US006564373B1

(12) **United States Patent**
Hughes et al.

(10) **Patent No.: US 6,564,373 B1**
(45) **Date of Patent: May 13, 2003**

(54) **INSTRUCTION EXECUTION MECHANISM**

(75) Inventors: **Kevin Hughes**, Glossop (GB); **Martin Pixton**, Sandbach (GB)

(73) Assignee: **International Computers Limited**, London (GB)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/522,579**

(22) Filed: **Mar. 10, 2000**

(30) **Foreign Application Priority Data**

Mar. 24, 1999 (GB) 9906652

(51) **Int. Cl.**[7] **G06F 9/45**

(52) **U.S. Cl.** **717/158**; 717/127; 717/159; 712/227

(58) **Field of Search** 712/209, 227; 717/151-153, 158, 159

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,428,786 A	*	6/1995	Sites	717/151
5,507,030 A	*	4/1996	Sites	717/136
5,590,329 A	*	12/1996	Goodnow et al.	717/144
5,613,118 A	*	3/1997	Heisch et al.	717/158
5,721,927 A		2/1998	Baraz	395/705
5,751,982 A		5/1998	Morley	395/800
5,894,576 A	*	4/1999	Bharadwaj	717/156
5,950,009 A	*	9/1999	Bortnikov et al.	717/158
6,044,220 A	*	3/2000	Breternitz, Jr.	717/139
6,192,513 B1	*	2/2001	Subrahmanyam	717/155

FOREIGN PATENT DOCUMENTS

EP	0 927 929	7/1999
EP	0 930 572	7/1999
WO	WO 92/15937	9/1992

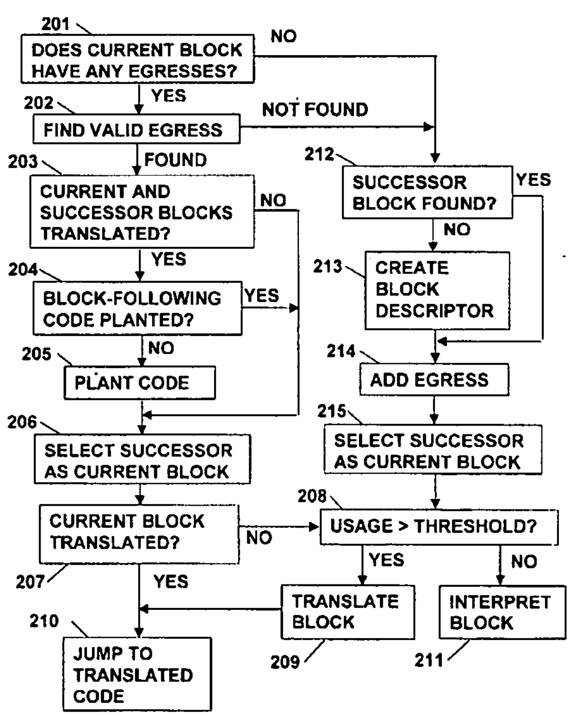
* cited by examiner

Primary Examiner—Gregory Morse
Assistant Examiner—Kenneth A Gross
(74) *Attorney, Agent, or Firm*—Barnes & Thornburg

(57) **ABSTRACT**

On completion of execution of a current block of instructions, a block completion process searches for potential successor blocks, using block descriptors and egress data structures. For each potential successor block, the process compares a set of entry conditions associated with the block with the exit conditions of the current block and, if a match is found, selects the potential successor block as the current block and executes it. A consistency check is performed, to compare the block identity of the successor block with an expected block identity. Block-following code is selectively planted into translated blocks, to call a successor block directly, by-passing the block completion process. The block-following code is optimised, in that it contains tests for entry conditions only if the results of those tests are not known at the time the block-following code is planted.

10 Claims, 2 Drawing Sheets



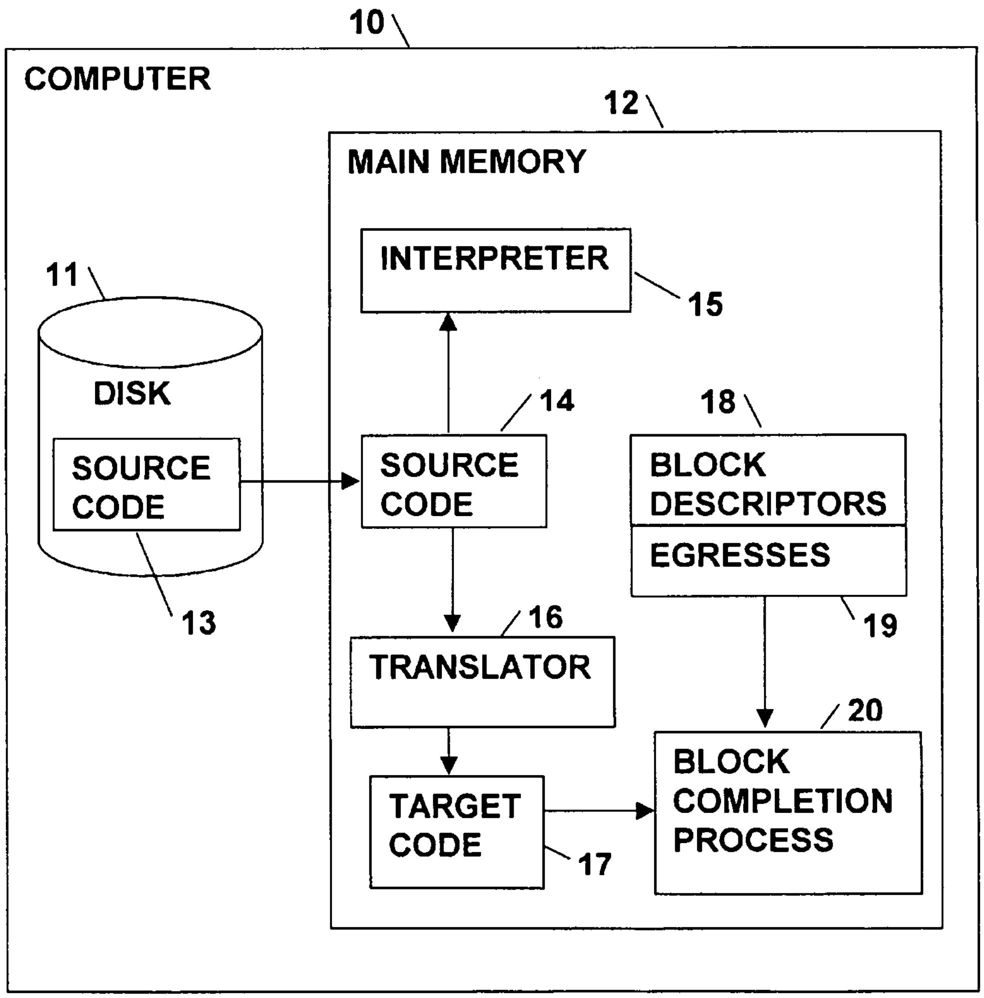

10
COMPUTER
12
MAIN MEMORY
11
DISK
SOURCE CODE
13
INTERPRETER
15
14
SOURCE CODE
18
BLOCK DESCRIPTORS
EGRESSES
19
16
TRANSLATOR
20
TARGET CODE
17
BLOCK COMPLETION PROCESS

FIG. 1

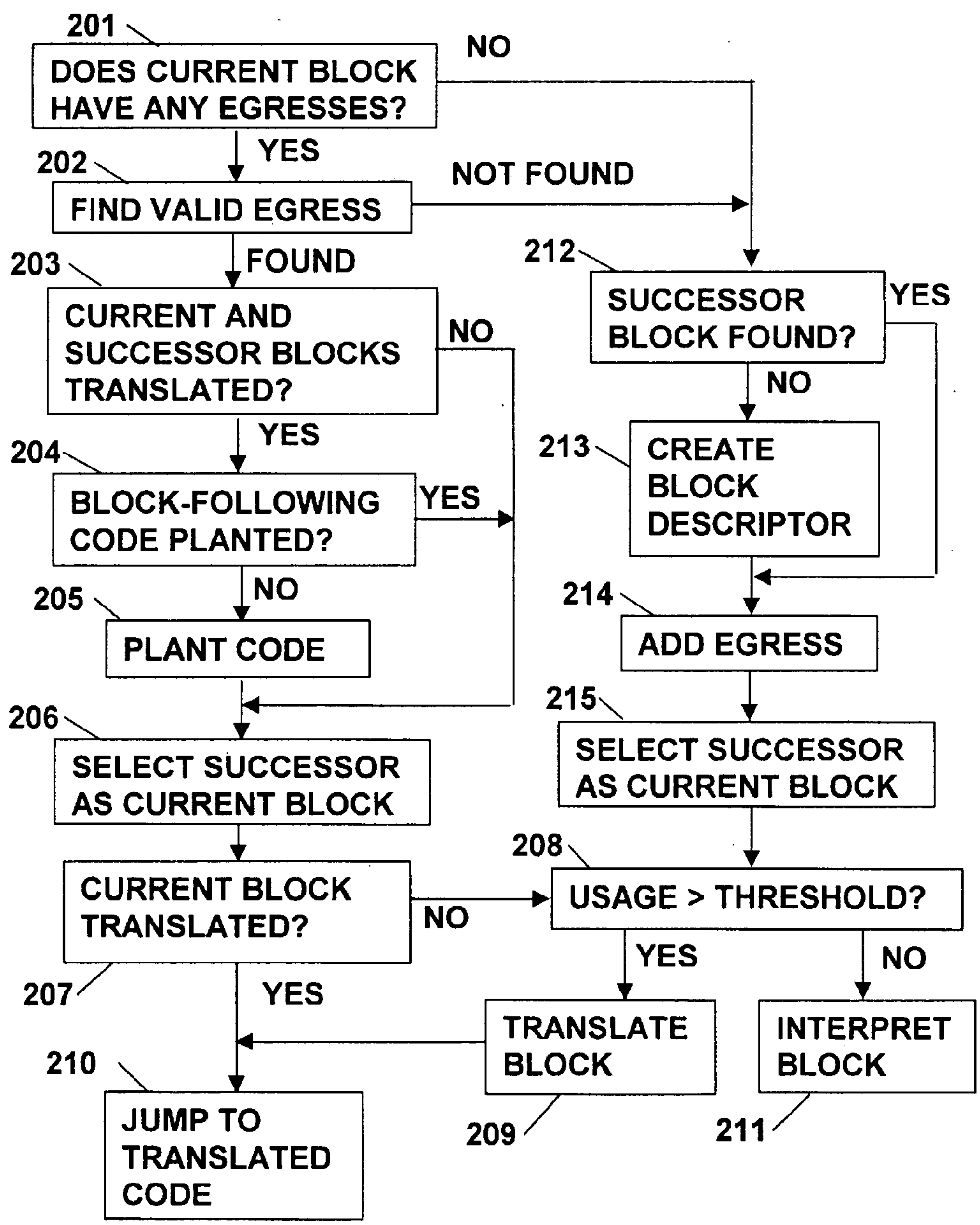

201
DOES CURRENT BLOCK HAVE ANY EGRESSES?
NO
YES
202
FIND VALID EGRESS
NOT FOUND
FOUND
203
CURRENT AND SUCCESSOR BLOCKS TRANSLATED?
NO
YES
204
BLOCK-FOLLOWING CODE PLANTED?
YES
NO
205
PLANT CODE
206
SELECT SUCCESSOR AS CURRENT BLOCK
CURRENT BLOCK TRANSLATED?
NO
207
YES
210
JUMP TO TRANSLATED CODE
212
SUCCESSOR BLOCK FOUND?
YES
NO
213
CREATE BLOCK DESCRIPTOR
214
ADD EGRESS
215
SELECT SUCCESSOR AS CURRENT BLOCK
208
USAGE > THRESHOLD?
YES
NO
TRANSLATE BLOCK
209
INTERPRET BLOCK
211

FIG.2

INSTRUCTION EXECUTION MECHANISM

BACKGROUND TO THE INVENTION

This invention relates to a mechanism for executing instructions in a computer system.

The invention is particularly concerned with a computer system in which source code instructions are translated into target code instructions for execution on a particular processor. This may be required, for example, where one processor is being used to emulate another, in which case the instructions for the processor being emulated must be translated into instructions for the emulating processor.

One approach, referred to as interpretation, is to create a software model of the instruction set of the processor being emulated. This model operates by reading each target instruction, decoding it, and selecting one of a number of sequences that perform the same function as the instruction being emulated. This fetch/decode/execute sequence is repeated for each source code instruction in turn.

A more efficient approach is to translate a block of source code instructions, rather than a single instruction. That is, the source code is divided into blocks, and each source code block is translated into a block of target code instructions, functionally equivalent to the source code block. Typically, a block has a single entry point and one or more exit points. The entry point is the target of a source code jump, while the (or each) exit is a source code jump.

Translating blocks is potentially much more efficient, since it provides opportunities for eliminating redundant instructions within the target code block, and other optimisations. Known optimising compiler techniques may be employed for this purpose. To increase efficiency further, the target code blocks may be held in main memory and/or a cache store, so that they are available for re-use if the same section of code is executed again, without the need to translate the block.

The present invention is concerned with the problem of sequencing the execution of such blocks, to ensure that each block is followed by an appropriate successor block. The object of the present invention is to provide an improved method for sequencing execution of code blocks.

SUMMARY OF THE INVENTION

According to the invention, a method of executing instructions in a computer system, comprises:

(a) storing a plurality of blocks of instructions, each block having a specified set of entry conditions associated with it;

(b) on completion of execution of a current block of instructions, searching for potential successor blocks; and

(c) for each potential successor block, comparing the set of entry conditions associated with that block with the exit conditions of the current block of instructions and, if a match is found, selecting the potential successor block as the current block and executing it.

It will be seen that the invention permits the use of multiple blocks for different entry conditions. The use of such multiple blocks is advantageous, in that each block can be tailored to its specific entry conditions, thereby improving its efficiency of execution. A preferred form of the invention also allows block-following code to be planted in the blocks, to optimise the sequencing between the blocks.

One embodiment of the invention will now be described by way of example with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram showing an instruction execution mechanism embodying the invention.

FIG. 2 is a flow chart of a block completion process.

DESCRIPTION OF AN EMBODIMENT OF THE INVENTION

FIG. 1 shows a host computer system **10**, including a disk memory **11**, and a main memory **12**.

The disk memory holds a number of source code blocks **13**. As will be described, these blocks are copied (paged) into the main memory on demand. Therefore, at any given time, the main memory holds a number of source code blocks **14**, corresponding to a subset of the blocks **13** in the disk memory.

The source code blocks consist of sequences of instructions in a source instruction set, which in this example is assumed to be the ICL VME instruction set. Each of the source code blocks has a set of entry conditions associated with it, these entry conditions consisting of:

The program counter (PC) value at entry to the block.

The accumulator size (ACS) value at entry to the block.

It should be noted that a plurality of source code blocks may exist with the same PC, each having a different ACS value.

An interpreter program **15** is provided for executing the source code blocks **14** in the cache. The interpreter takes each source code instruction in turn, decodes it, and selects one of a number of target code sequences in the instruction set of the host computer that perform the same function as the source code instruction. Details of the interpreter are not relevant to the present invention, and so it will not be described in any further detail.

A translator program **16** is also provided. This can be called, as will be described, to translate a selected source code block **14** into a target code block **17**, in the instruction set of the host computer. The translation takes account of the current entry conditions for the block, so that each translated block is specific to a particular set of entry conditions. Details of the translator are not relevant to the present invention, and so it will not be described in any further detail.

Each source code block may have a block descriptor **18** associated with it. Each block descriptor contains the following information:

BID An identity number, uniquely identifying this block descriptor.

BACS The accumulator size (ACS) on entry into the block.

BPC The program counter (PC) value on entry into the block.

BNIB The address of the translated target code (if any) corresponding to the source code block.

USAGE A count of the number of times the block has been executed.

When a new block descriptor is required, if there is no space in memory for creating a new block descriptor, an existing block descriptor may be reused.

Every time a block descriptor is created or reused, it is allocated a new block identifier BID equal to a global block identifier value GLOBAL_BID, and GLOBAL_BID is then incremented. This ensures that each block has a unique BID value. (When GLOBAL_BID eventually reaches its maximum value, it will be reset to zero, and in this case all

existing block descriptors and target code blocks must be flushed from the cache. However, this will happen only rarely).

The way in which the code blocks are threaded together is specified by data structures **19**, referred to herein as egresses, linked to the block descriptors **18**. Each block descriptor may have one or more egresses linked to it. Each egress contains information relating to a potential successor block for the current block, as follows:

BPC The program counter (PC) value on entry into the successor block.

BID The identity number of the block descriptor associated with the successor block.

BLK The address of the block descriptor associated with the successor block.

BOPT A flag, indicating whether or not block-following code has been planted into the target code for this egress.

In general, whenever execution of a code block is completed (either execution of source code by the interpreter, or execution of a translated target code) a block completion process **20** is executed, to select the appropriate successor block. The exception to this is the case where a translated block contains block-following code. In that case, the block-following code may cause a jump to the required successor block, directly from the translated code, without executing the block completion process, as will be described.

The block completion process **20** will now be described with reference to FIG. **2**.

(Step **201**) The process first accesses the block descriptor **18** associated with the current code block (i.e. the code block that has just completed), and checks whether this descriptor has any egresses **19** linked to it.

(Step **202**) If there are one or more egresses linked to the current block, the process selects each egress in turn, and performs a test to check whether this is a valid egress for the current conditions. This test can be expressed as follows:

```
current_block.egress.BLK != NULL                 AND
PC == current_block.egress.BPC                   AND
ACS == current_block.egress.BLK->BACS            AND
Current_block.egress.BID == current block.egress.BLK->BID
```

The first line of this test checks that the BLK value in the egress is not null, i.e. the egress points to a valid successor block. The second line checks that the current program counter value (PC) is equal to the BPC value in the egress. The third line checks that the current accumulator size (ACS) is equal to the BACS value in the descriptor pointed to by the BLK value in the egress. The second and third lines therefore check that the entry conditions for the successor block are the same as the conditions on exit from the current block. The fourth line is a consistency check, which confirms that the BID value in the egress equals the BID value in the descriptor pointed to by the BLK value in the egress. This consistency check is necessary to ensure that the block pointed to by the egress is the same block as when the egress was created, i.e. the block descriptor has not been re-used. The egress is considered valid only if all four of these conditions are true.

(Step **203**) If a valid egress is found in Step **202**, the process performs a test to check whether both the current block and the successor block have been translated:

```
current_block.egress.BLK->BNIB!=NULL
AND
current_block.BNIB!=NULL
```

The first line of this test checks that the BNIB value in the descriptor pointed to by the BLK value in the egress is not null. The second line checks that the BNIB value in the current block descriptor is not null. If both of these conditions are true, the process continues at Step **204**; otherwise it goes to Step **206**.

(Step **204**) The process checks the BOPT flag in the selected egress to determine whether or not block-following code has been planted for this particular block-to-block link. If BOPT is False, the process continues at Step **205**; otherwise it goes to Step **206**.

(Step **205**) The process plants block-following code into the current translated block, to link the current block to its successor. This code will be performed next time this block is executed.

(Step **206**) The process then performs the action:

```
Current_block=current_block.egress.BLK.
```

This designates the successor block (i.e. the block indicated by the BLK value in the egress) as the new current block.

(Step **207**) The process then checks the BNIB value in the current block descriptor. If it is null, this indicates that the block has not been translated, and the process continues at Step **208**. Otherwise, it continues at Step **210**.

(Step **208**) The process checks whether the USAGE count in the current block descriptor is greater than a predetermined threshold value, indicating that the (as yet untranslated) block has been executed more than a certain number of times. If the USAGE count is greater than the threshold, the process continues at Step **209**; otherwise, it proceeds to Step **211**.

(Step **209**) The process calls the translator **16**, to translate the block. The translated code is stored in the main memory, and the BNIB value in the current block descriptor is set equal to the address of the translated code.

The translated code has two entry points: a first entry point which is used when entry is made from the block completion process, and a second entry point for use when entry is made from block-following code.

(Step **210**) A jump is then made to the first entry point of the translated code, and this code is executed.

(Step **211**) If the USAGE count is not greater than the threshold, the process calls the interpreter **15**, to execute the source code block. The USAGE count is incremented.

(Step **212**) If it was found at Step **201** that the current block does not have any egresses, or if no valid egresses were found at Step **202**, the process searches through the block descriptors **14**, looking for a successor block whose entry conditions match the exit conditions (PC, ACS) of the current block.

(Step **213**) If the required successor block is not found in the current set of block descriptors, then the source code **13**,**14** is explored to find the required successor block, i.e. to find the source code block whose entry conditions match the exit conditions (PC, ACS) of the current block, and a new block descriptor **18** is created to describe it. In the new block descriptor, BACS and BPC are set equal to the values of ACS and PC on exit from the current block, BNIB is set to NULL, and USAGE is set to 0.

(Step **214**) A new egress, pointing to the successor block, is then created and linked to the current block descriptor. The BOPT flag in the new egress is set to FALSE.

(Step 215) The process then designates the successor block as the new current block. Operation continues at Step 208 as described above.

In summary, it can be seen that on completion of a block, the block completion process searches for potential successor blocks. For each potential successor, the entry conditions (PC, ACS) associated with that block are compared with the conditions on exit from the current block. If they match, the successor block becomes the new current block, and a jump is made to it.

The block-following code planted at Step 205 above will now be described. The block-following code planted depends on whether the values of PC and ACS on exit from the block are predictable at the time the block-following code is planted, or can be determined only at run time. There are four cases, as follows.

Case 1: both PC and ACS are predictable. In this case, the block-following code includes an unconditional jump to the translated code for the successor block, as follows:

```
JUMP current_block.egress.BLK->BNIB.
```

Case 2: PC not predictable, ACS predictable. In this case, the block-following code includes the following:

```
IF current_block.egress.PC==PC
THEN
  JUMP current_block.egress.BLK->BNIB
END IF
```

In other words, a jump is made to the translated code for the successor block, only if the PC value on entry to the successor block is equal to the current PC value.

Case 3: PC predictable, ACS not predictable. In this case, the block-following code includes the following:

```
IF current_block.egress.BLK->BACS==ACS
THEN
  JUMP current_block.egress.BLK->BNIB
END IF
```

In other words, a jump is made to the translated code for the successor block, only if the ACS value on entry to the successor block is equal to the current ACS value.

Case 4: both PC and ACS are unpredictable. In this case, the block-following code includes the following:

```
IF current_block.egress.PC==PC AND
current_block.egress.BLK->BACS==ACS
THEN
  JUMP current_block.egress.BLK->BNIB
END IF
```

In other words, a jump is made to the translated code for the successor block, only if both the PC and ACS value on entry to the successor block are equal to the current PC and ACS values.

Thus, the block-following code is optimised, in that it contains tests for entry conditions only if the results of those tests are unknown or unpredictable at the time the block-following code is planted.

The values current_block.egress.PC, current_block.egress.BLK and current_block.egress.BLK->BACS are planted as literal values, and hence do not require access to the egress data structure.

If all the tests in the block-following code are successful, a jump will be made to the translated code for the successor block without having to call the block completion process. Otherwise, the block completion process 20 will be called.

As mentioned above, each translated target code block 17 has two entry points: a first entry point which is used when entry is made from the block completion process 20, and a second entry point for use when entry is made from the block-following code.

The second entry point performs the following consistency check:

```
IF parent_block.BID!=current_block.egress.BLK->BID
THEN
  Return to block completion process
END IF
Current_block=current_block.egress.BLK
```

where parent_block is the block from which this code was generated.

In this consistency check, parent_block.BID is planted as a literal value, and hence does not require access to the egress data structure.

This consistency check confirms that the block identifier of the block being entered is equal to the expected block identifier. If so, this block is designated as the new current block, and execution of the translated code continues. Otherwise, the block completion process 20 is returned to.

It can be seen that planting the block-following code improves performance, because it by-passes the standard block completion process 20 in cases where it is known that a valid translated successor block exists, with the correct entry conditions. Performance is also improved because the block-following code contains literal values, and so does not require to access the egress data structures.

SOME POSSIBLE MODIFICATIONS

It will be appreciated that many modifications may be made to the system described above without departing from the scope of the present invention. For example, entry conditions other than (or additional to) PC and ACS may be tested for.

What is claimed is:

1. A method of executing instructions in a computer system, the method comprising:

(a) storing a plurality of blocks of instructions, each block having a specified set of entry conditions associated with it;

(b) executing a current block of instructions to produce a set of run-time exit conditions;

(c) on completion of execution of said current block of instructions, searching for potential successor blocks; and

(d) for each potential successor block, comparing the set of entry conditions associated with that block with the run-time exit conditions of the current block of instructions and, if a match is found, selecting the potential successor block as the current block and executing it.

2. A method according to claim 1, further including selectively planting executable block-following code into the current block of instructions on completion of execution of said current block of instructions, said executable block-following code including instructions to call a successor block directly when said current block of instructions is next executed.

3. A method according to claim 2 wherein the planted block-following code is optimised, in that it contains tests for entry conditions only if the results of those tests are not known at the time the block-following code is planted.

4. A method according to claim 1 including

(a) creating a number of block descriptors, each of which holds information relating to a particular block; and

(b) creating a number of egress data structures, each associated with one of the block descriptors, each egress pointing to a potential successor block; and

(c) accessing the egress data structure for the potential successor block to obtain the entry conditions for that block.

5. A method of executing instructions in a computer system, the method comprising:

(a) storing a plurality of blocks of instructions, each block having a specified set of entry conditions associated with it; and

(b) planting executable block-following code into at least some of said blocks, and

(c) executing a current block of instructions to produce a set of run-time exit conditions for said current block;

(d) said executable block-following code including tests to determine whether the run-time exit conditions for the current block match the entry conditions for a specified successor block;

(e) wherein if said tests are satisfied the executable block-following code causes a jump directly to the specified successor block.

6. A method according to claim 5 wherein the block-following code is optimised, in that it contains tests for exit conditions only if those exit conditions are unpredictable at the time the block-following code is planted.

7. A method according to claim 5 wherein, if said tests are not successful, the block-following code calls a block-completion process to search for a successor block whose entry conditions match the exit conditions of the current block.

8. A method according to claim 5 wherein, on entry from the block-following code of a predecessor block, the successor block performs a consistency check to confirm that it is the correct successor block and, if this consistency check is unsuccessful, the successor block calls a block-completion process to search for a successor block whose entry conditions match the exit conditions of the predecessor block.

9. A method according to claim 5 wherein the block-following code is planted into a block only if that block and its successor block have both been translated into the native instruction code of the computer system.

10. A method of executing instructions in a computer system, the method comprising:

(a) storing a plurality of blocks of instructions;

(b) storing a plurality of data structures holding information on entry conditions and potential successor blocks for each of the blocks of instructions;

(c) executing a current block of instructions to produce a set of run-time exit conditions;

(d) on completion of execution of said current block of instructions, searching the data structures to locate a successor block whose entry conditions match the run-time exit conditions of the current block of instructions; and

(e) if a successor block whose entry conditions match the run-time exit conditions of the current block of instructions is found, selecting that successor block as the current block and executing it.

* * * * *

US006085029A

United States Patent [19]

Kolawa et al.

[11] **Patent Number: 6,085,029**

[45] **Date of Patent: Jul. 4, 2000**

[54] **METHOD USING A COMPUTER FOR AUTOMATICALLY INSTRUMENTING A COMPUTER PROGRAM FOR DYNAMIC DEBUGGING**

[75] Inventors: **Adam K. Kolawa**, Sierra Madre, Calif.; **Roman Salvador**, Barcelona, Spain; **Wendell T. Hicken**, Whittier; **Bryan R. Strickland**, Los Angeles, both of Calif.

[73] Assignee: **Parasoft Corporation**, Monrovia, Calif.

[*] Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

[21] Appl. No.: **08/701,097**

[22] Filed: **Aug. 21, 1996**

Related U.S. Application Data

[63] Continuation of application No. 08/435,759, May 9, 1995, Pat. No. 5,581,696.

[51] **Int. Cl.**^{7} **G06F 11/00**; G01R 31/28

[52] **U.S. Cl.** **395/183.14**; 395/704; 395/708

[58] **Field of Search** 395/183.14, 500, 395/183.13, 183.01, 701–708; 364/267, 267.2, 267.91, 267.8, 268.1

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,620,303	10/1986	Tschoepe	395/183.01
5,175,856	12/1992	Van Dyke	395/700
5,270,712	12/1993	Iyer	341/50
5,371,747	12/1994	Brooks et al.	395/183.14
5,450,586	9/1995	Kuzara	395/700
5,465,321	11/1995	Smyth	706/20
5,475,588	12/1995	Schabes	364/419.08
5,493,678	2/1996	Arcuri	395/600
5,581,697	12/1996	Gramlich	395/183.11
5,583,988	12/1996	Crank	395/185.01
5,590,329	12/1996	Goodnow, II	395/708
5,671,416	9/1997	Elson	395/702
5,680,622	10/1997	Even	395/709

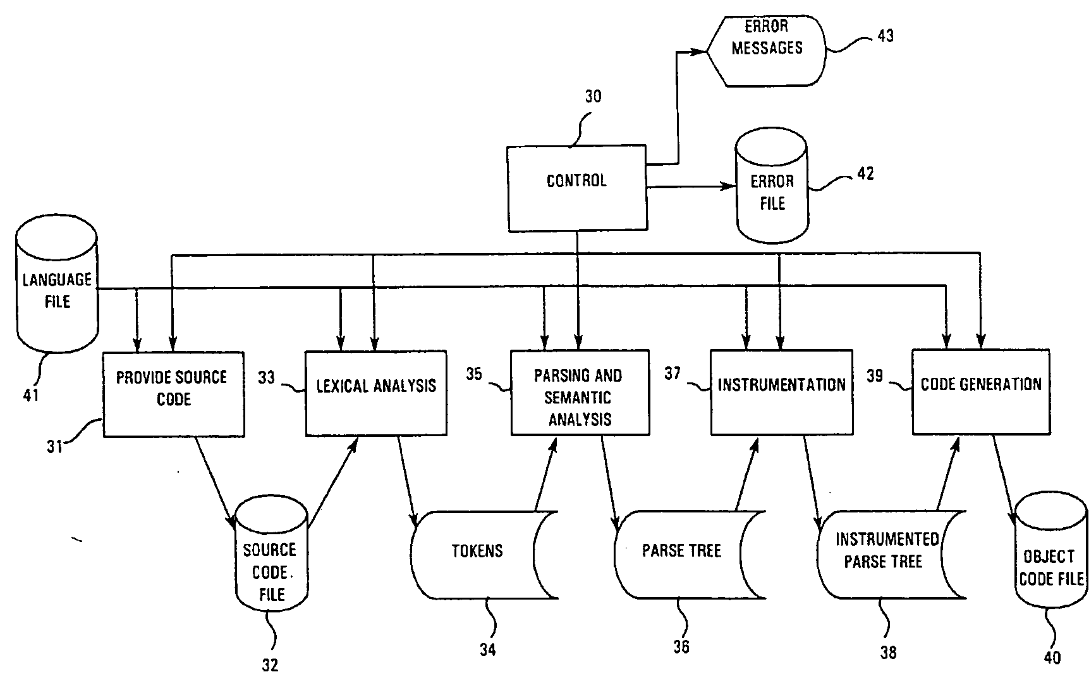
Primary Examiner—Dieu-Minh T. Le
Attorney, Agent, or Firm—Christie, Parker & Hale, LLP

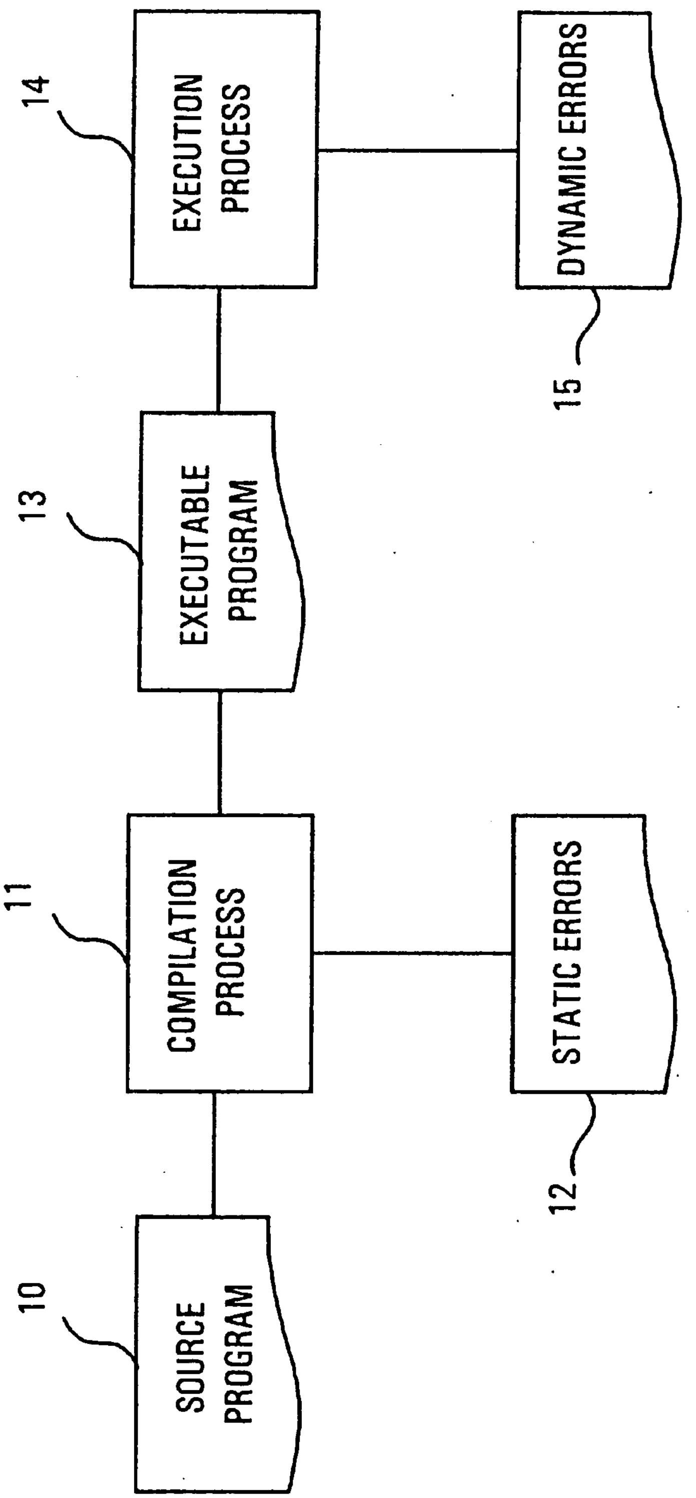
[57] **ABSTRACT**

A method for automatically instrumenting a computer program for dynamic debugging. Such a computer program comprising source code written in a programming language for executing instructions on the computer. The source code is provided as a sequence of statements in a storage device to the computer. Each of the statements are separated into tokens representing either an operator or at least one operand. A parse tree is built according to a set of rules using the set of tokens. The parse tree is instrumented to create an instrumented parse tree for indicating that an error condition occurred in the computer program during execution. Object code is generated from the instrumented parse tree and stored in a secondary storage device for later execution using an error-checking engine that indicates error conditions present in the computer program.

50 Claims, 32 Drawing Sheets

Microfiche Appendix Included
(17 Microfiche, 1585 Pages)





EXECUTION PROCESS
14
DYNAMIC ERRORS
15
EXECUTABLE PROGRAM
13
COMPILATION PROCESS
11
STATIC ERRORS
12
SOURCE PROGRAM
10

FIG. 1

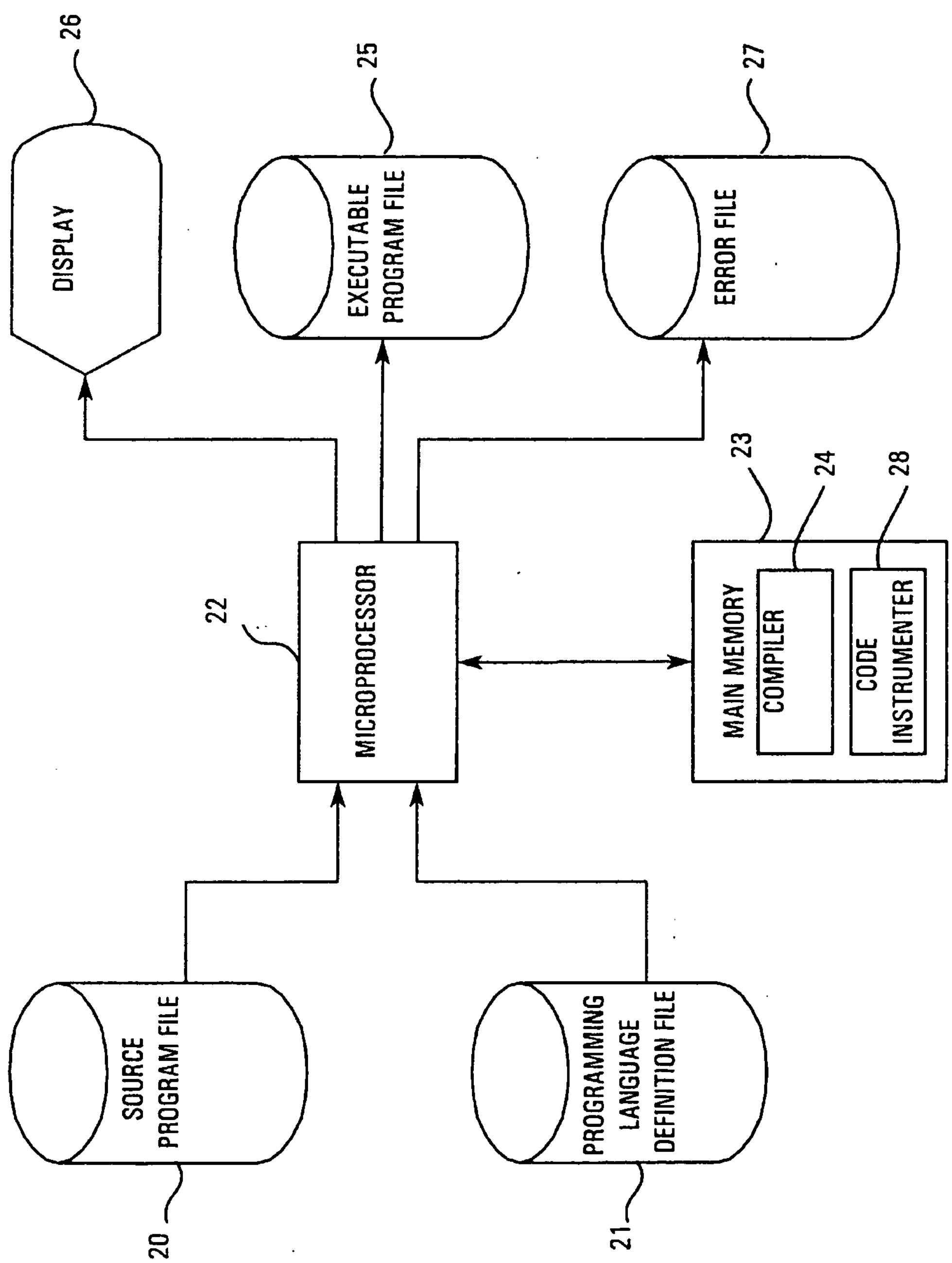

DISPLAY
26
EXECUTABLE PROGRAM FILE
25
ERROR FILE
27
MICROPROCESSOR
22
MAIN MEMORY
23
COMPILER
24
CODE INSTRUMENTER
28
SOURCE PROGRAM FILE
20
PROGRAMMING LANGUAGE DEFINITION FILE
21

FIG. 2

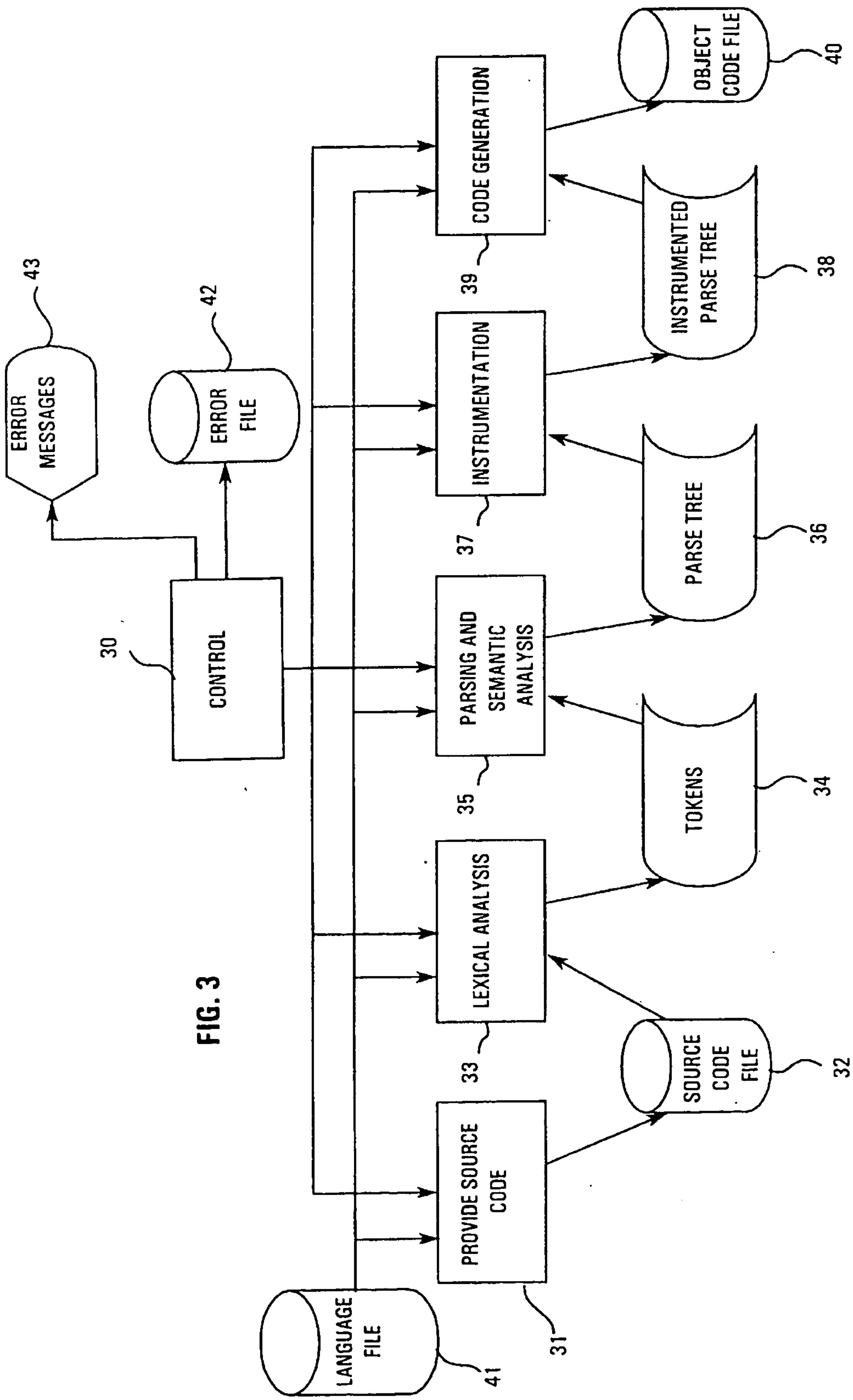

ERROR MESSAGES
43
ERROR FILE
42
CONTROL
30
CODE GENERATION
39
OBJECT CODE FILE
40
INSTRUMENTATION
37
INSTRUMENTED PARSE TREE
38
PARSING AND SEMANTIC ANALYSIS
35
PARSE TREE
36
LEXICAL ANALYSIS
33
TOKENS
34
PROVIDE SOURCE CODE
31
SOURCE CODE FILE
32
LANGUAGE FILE
41

FIG. 3

FIG. 4

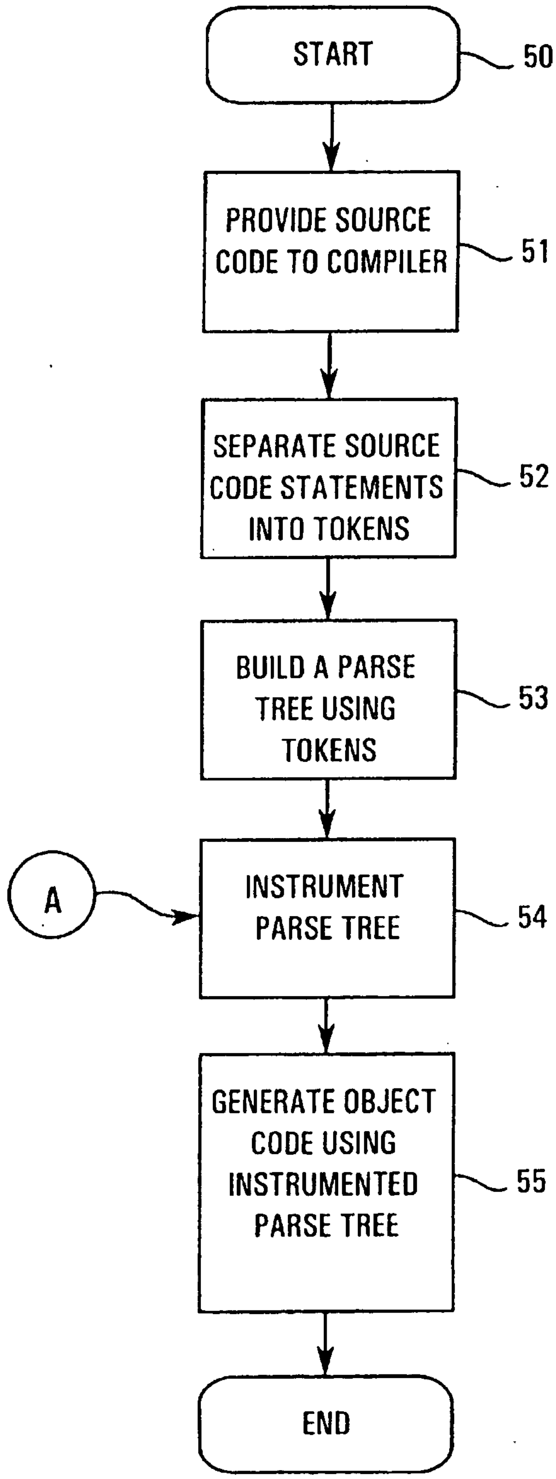
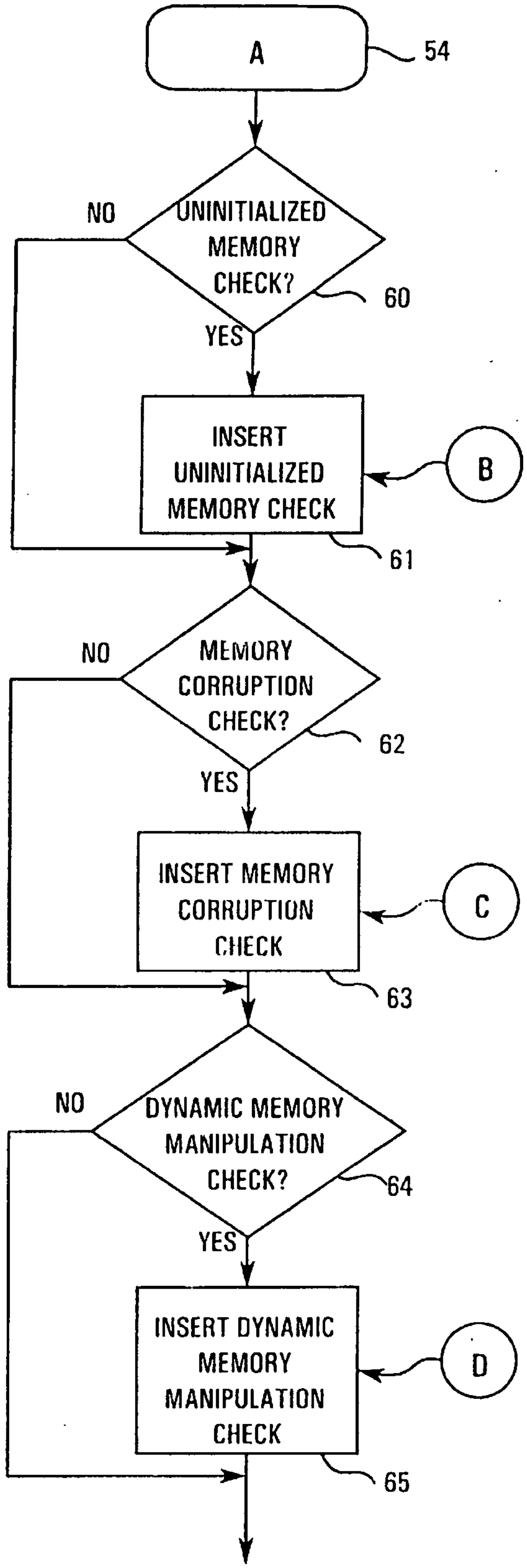

START
50
PROVIDE SOURCE CODE TO COMPILER
51
SEPARATE SOURCE CODE STATEMENTS INTO TOKENS
52
BUILD A PARSE TREE USING TOKENS
53
A
INSTRUMENT PARSE TREE
54
GENERATE OBJECT CODE USING INSTRUMENTED PARSE TREE
55
END

FIG. 5A



TO FIG. 5B

FIG. 5B

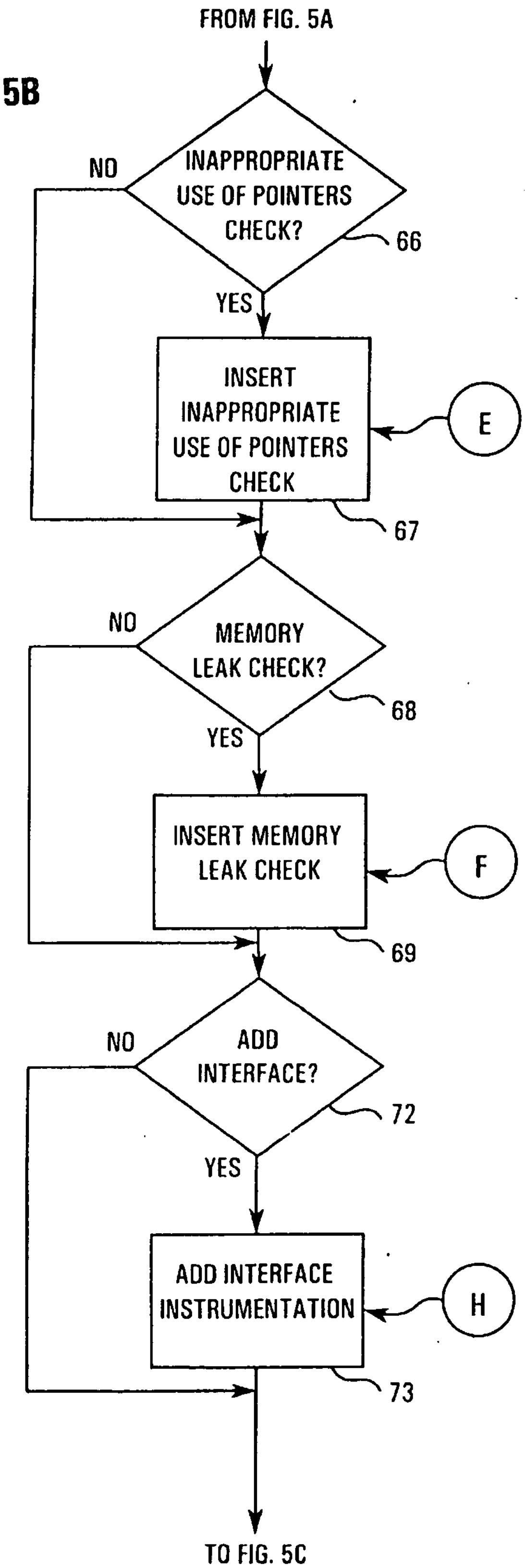

FROM FIG. 5A
INAPPROPRIATE USE OF POINTERS CHECK?
NO
YES
66
INSERT INAPPROPRIATE USE OF POINTERS CHECK
E
67
MEMORY LEAK CHECK?
NO
YES
68
INSERT MEMORY LEAK CHECK
F
69
ADD INTERFACE?
NO
YES
72
ADD INTERFACE INSTRUMENTATION
H
73
TO FIG. 5C

FIG. 5C

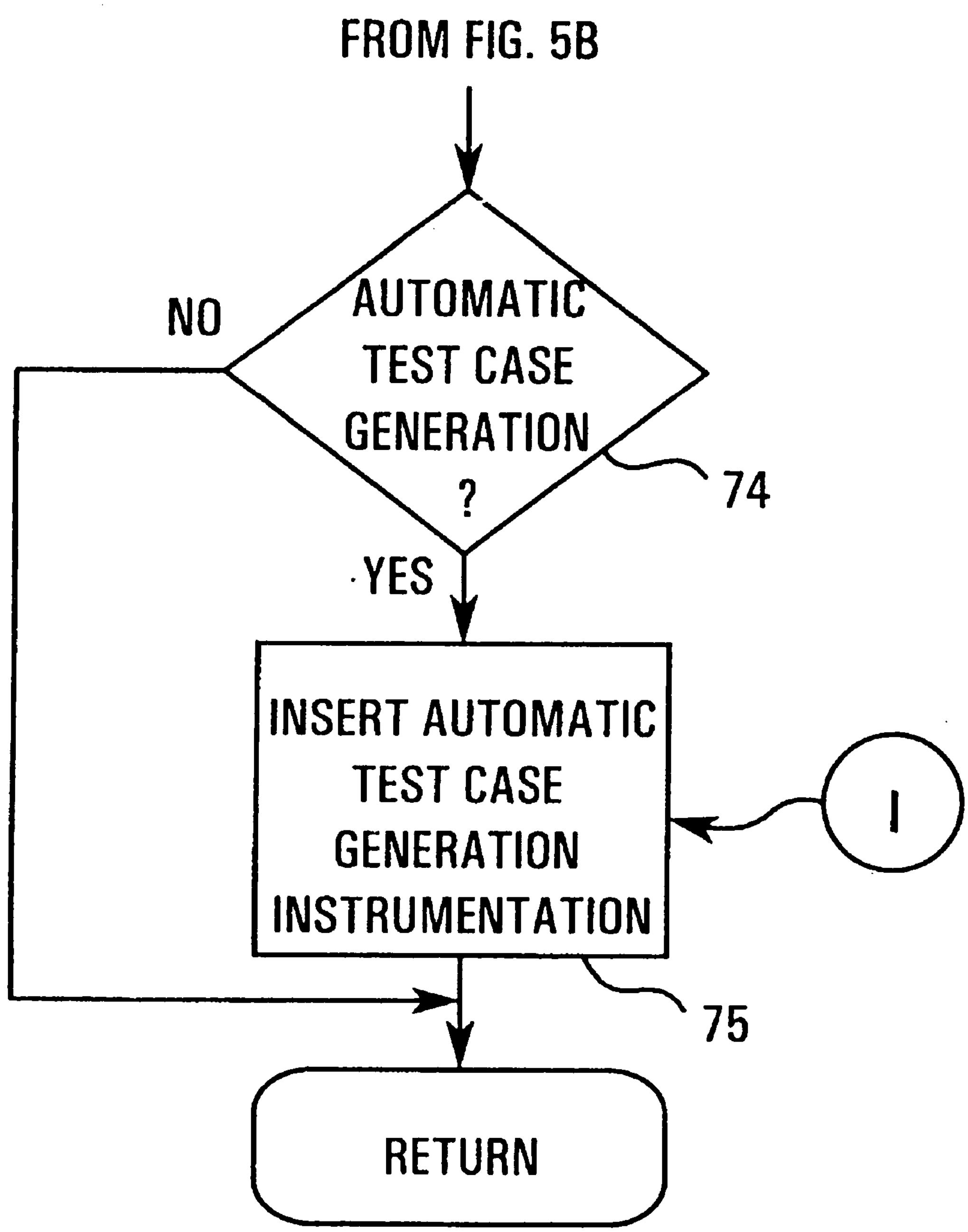

FROM FIG. 5B
NO
AUTOMATIC TEST CASE GENERATION ?
74
YES
INSERT AUTOMATIC TEST CASE GENERATION INSTRUMENTATION
I
75
RETURN

FIG. 6

```
int main()
{
  int a, i;
  a=i;
  return 0;
}
```

FIG. 10

```
int main()
{
  int i, A[10];
  for (i=1; i<=10; i++) {
      A[i]=0;
}
  return 0;
}
```

FIG. 14

```
int main()
{
  char *ptr;
  ptr=malloc(10);
  ptr++;
  free(ptr);
  return 0;
}
```

FIG. 19

```
long a, b, (*foo) ();
foo= (long (*) ())&a;
b= foo();
```

FIG. 7

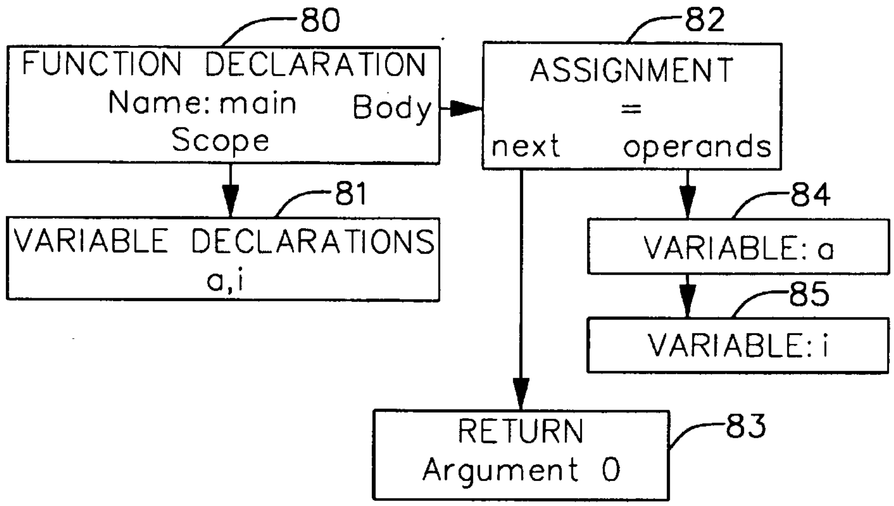

80
FUNCTION DECLARATION
Name: main
Body
Scope
82
ASSIGNMENT
=
next
operands
81
VARIABLE DECLARATIONS
a,i
84
VARIABLE: a
85
VARIABLE: i
RETURN
Argument 0
83

FIG. 9

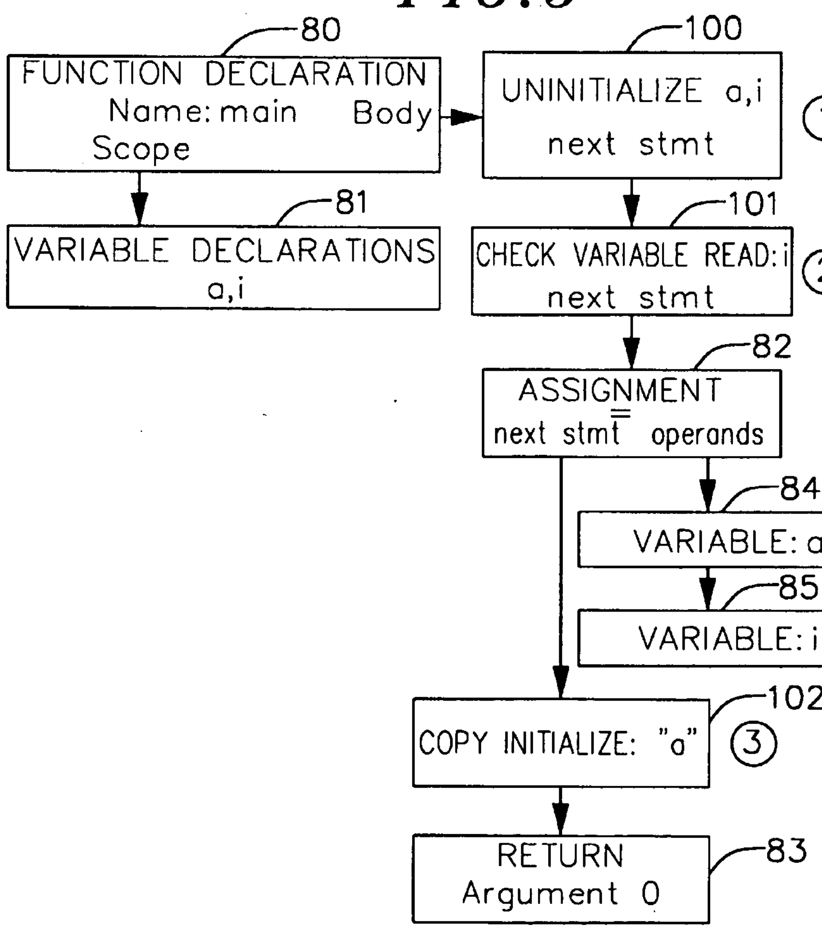
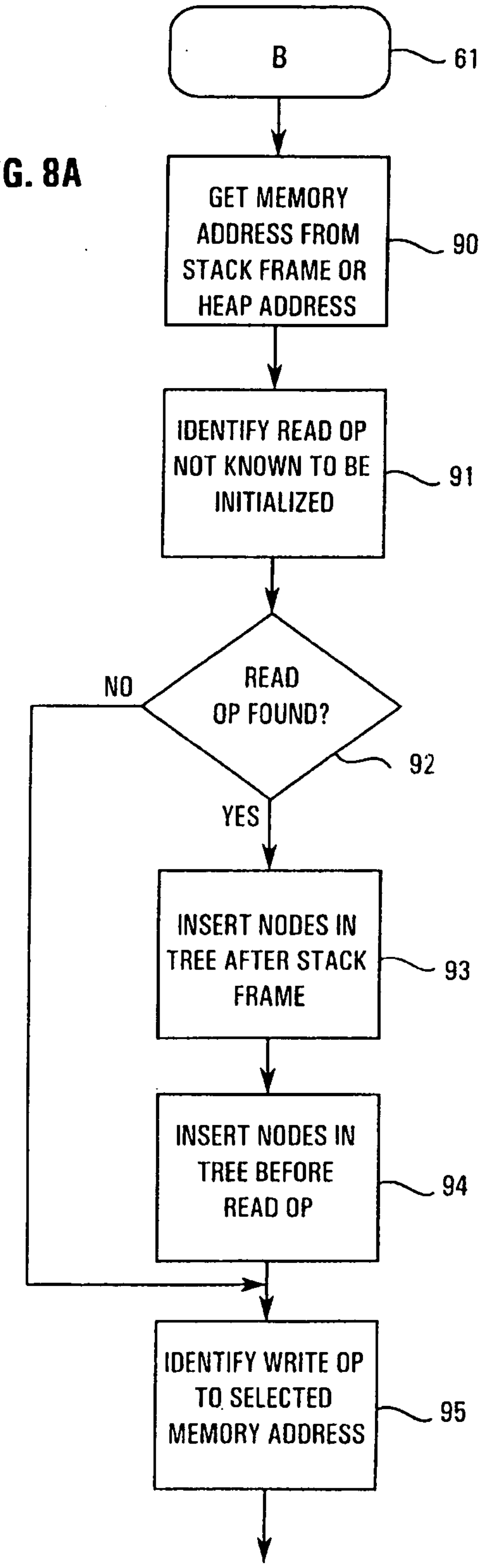

80
FUNCTION DECLARATION
Name: main
Body
Scope
100
UNINITIALIZE a,i
next stmt
1
81
VARIABLE DECLARATIONS
a,i
101
CHECK VARIABLE READ: i
next stmt
2
82
ASSIGNMENT
=
next stmt
operands
84
VARIABLE: a
85
VARIABLE: i
102
COPY INITIALIZE: "a"
3
RETURN
Argument 0
83

FIG. 8A



TO FIG. 8B

FIG. 8B

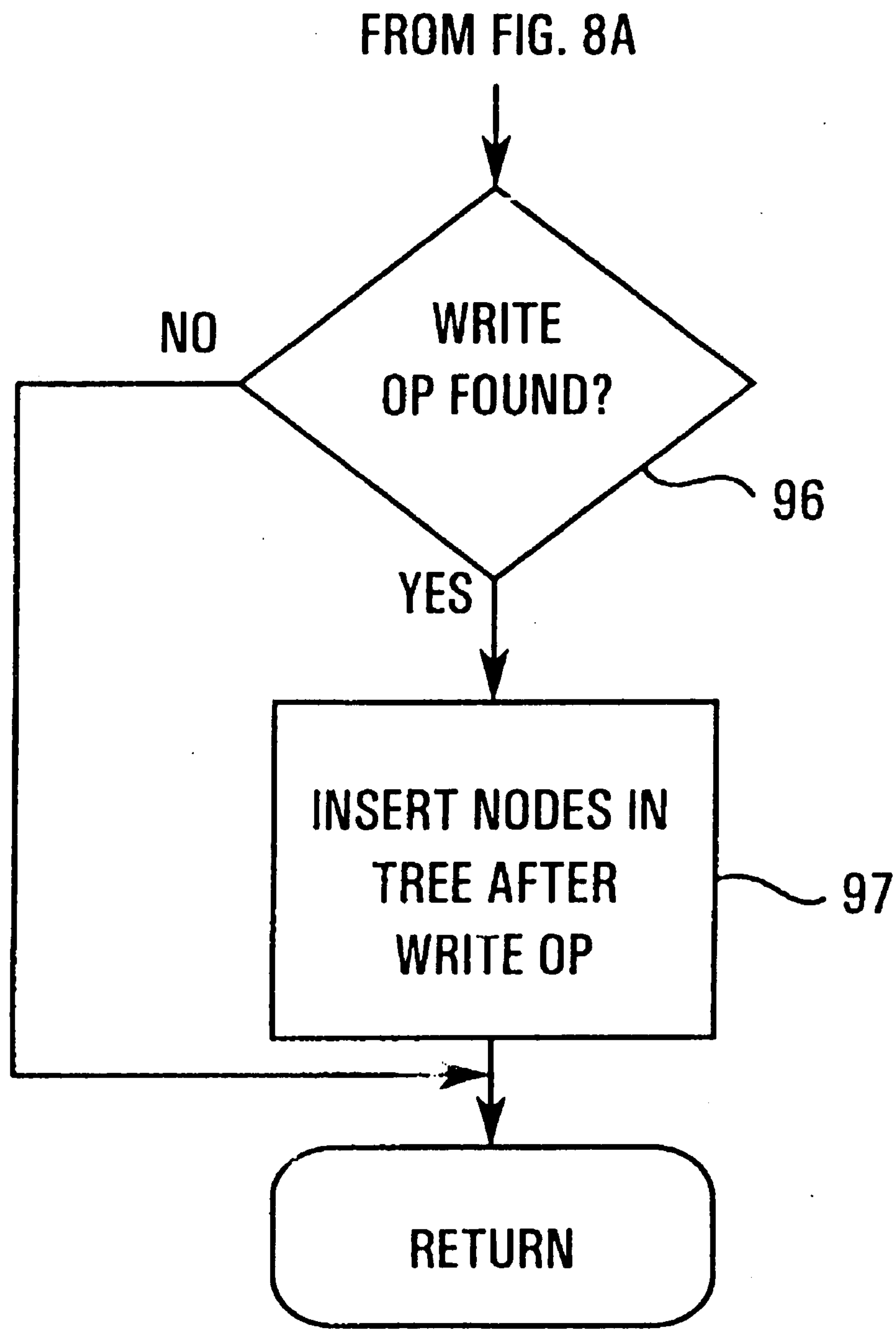

FROM FIG. 8A
NO
WRITE OP FOUND?
96
YES
INSERT NODES IN TREE AFTER WRITE OP
97
RETURN

FIG. 11

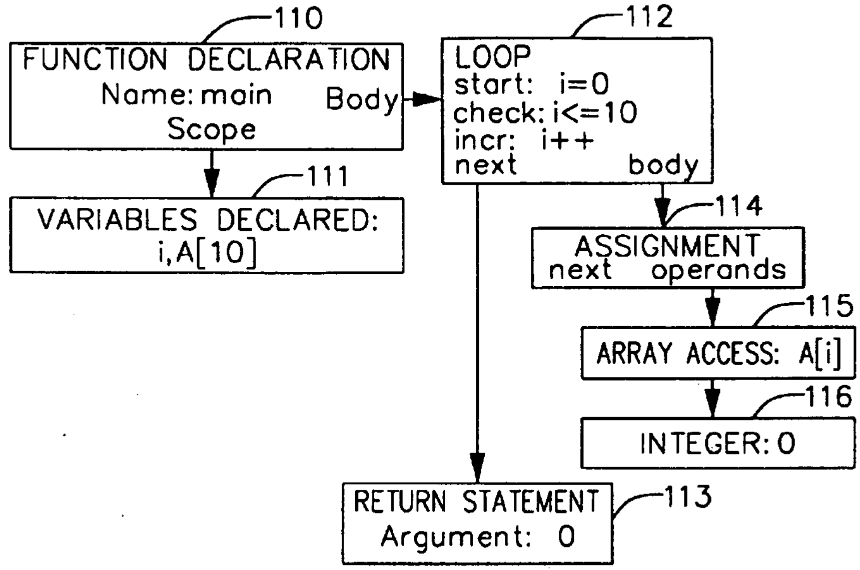
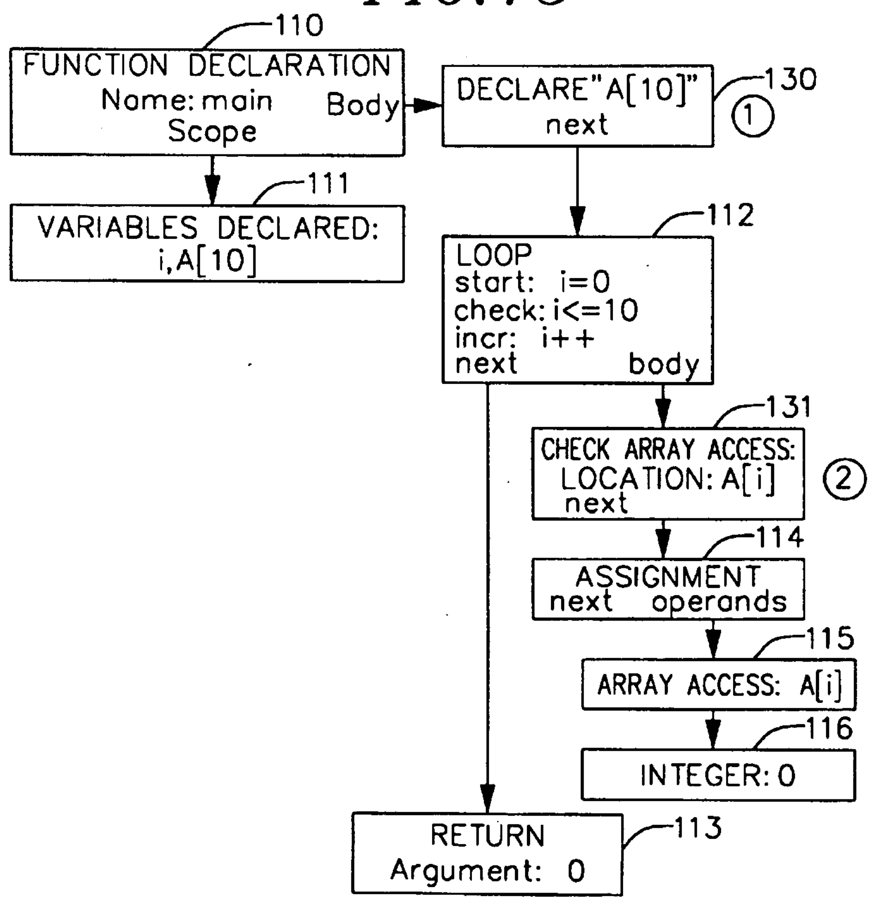

110
FUNCTION DECLARATION
Name: main
Body
Scope
111
VARIABLES DECLARED:
i,A[10]
112
LOOP
start: i=0
check: i<=10
incr: i++
next
body
114
ASSIGNMENT
next operands
115
ARRAY ACCESS: A[i]
116
INTEGER: 0
RETURN STATEMENT
Argument: 0
113

FIG. 13


110
FUNCTION DECLARATION
Name: main
Body
Scope
111
VARIABLES DECLARED:
i,A[10]
DECLARE "A[10]"
next
130
1
112
LOOP
start: i=0
check: i<=10
incr: i++
next
body
131
CHECK ARRAY ACCESS:
LOCATION: A[i]
next
2
114
ASSIGNMENT
next operands
115
ARRAY ACCESS: A[i]
116
INTEGER: 0
RETURN
Argument: 0
113

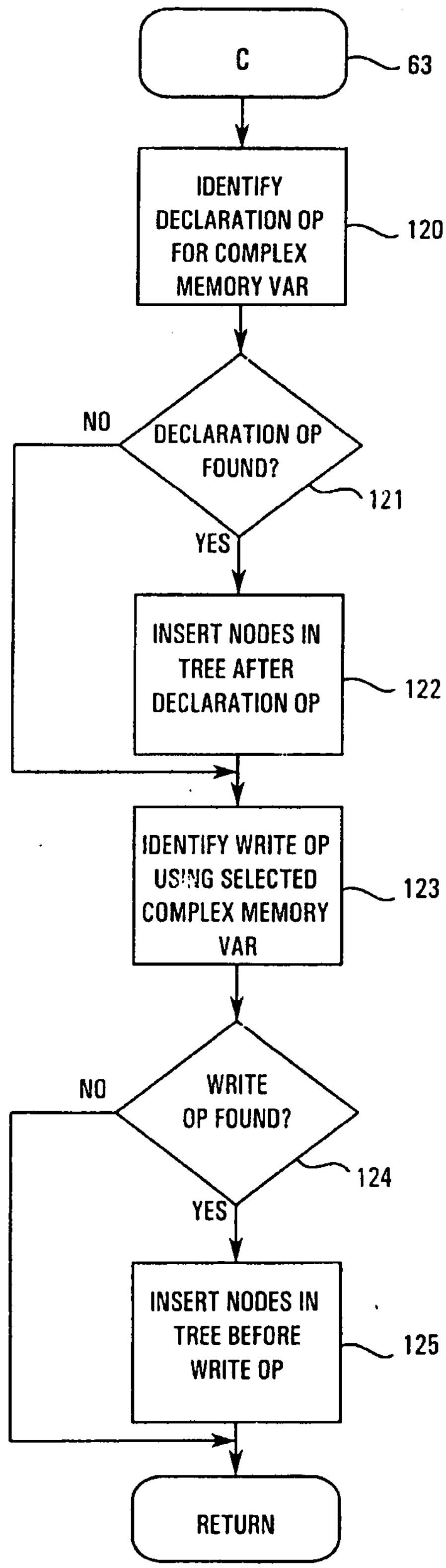

C
63
IDENTIFY DECLARATION OP FOR COMPLEX MEMORY VAR
120
NO
DECLARATION OP FOUND?
121
YES
INSERT NODES IN TREE AFTER DECLARATION OP
122
IDENTIFY WRITE OP USING SELECTED COMPLEX MEMORY VAR
123
NO
WRITE OP FOUND?
124
YES
INSERT NODES IN TREE BEFORE WRITE OP
125
RETURN

FIG. 12

FIG. 15

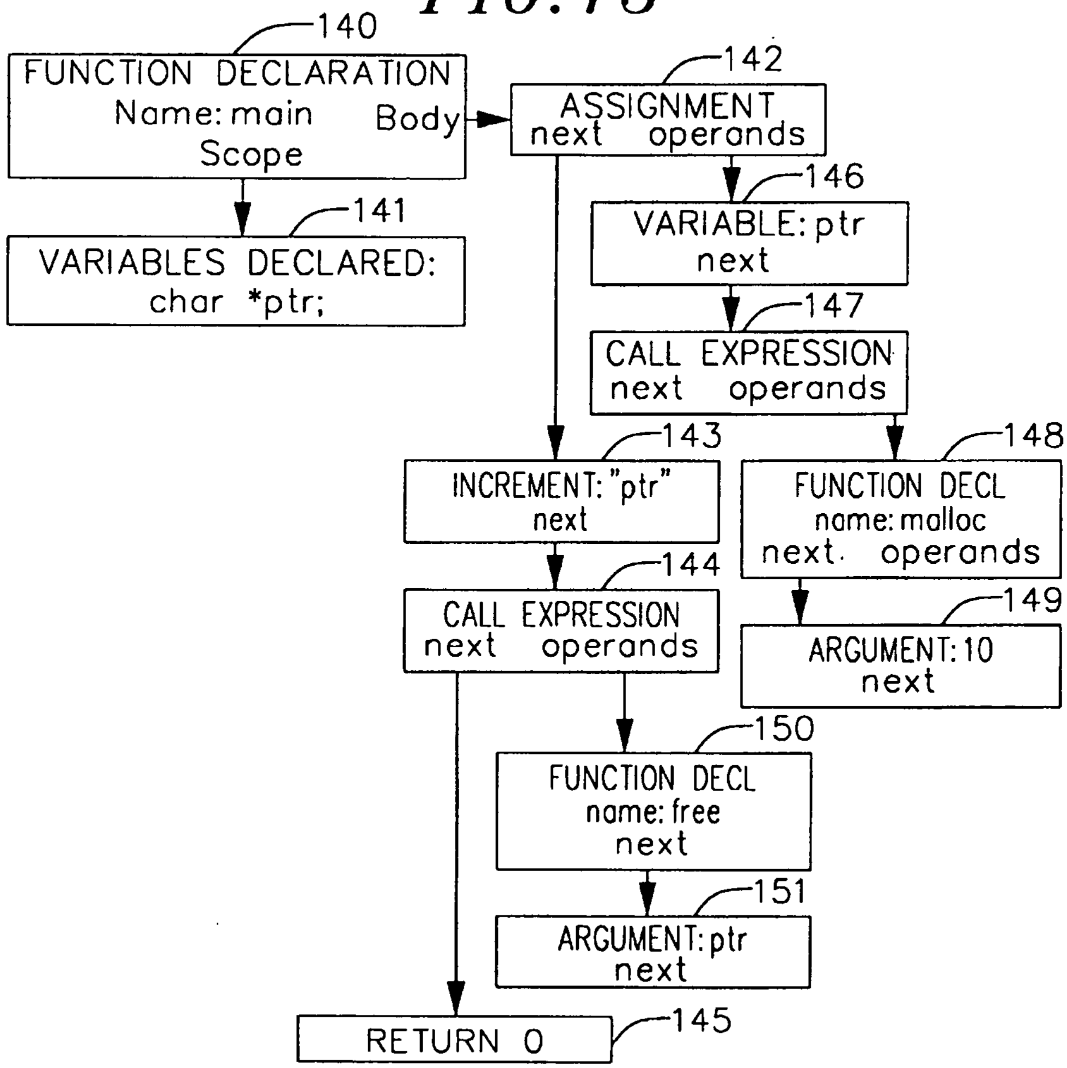

140
FUNCTION DECLARATION
Name: main
Body
Scope
141
VARIABLES DECLARED:
char *ptr;
142
ASSIGNMENT
next operands
146
VARIABLE: ptr
next
147
CALL EXPRESSION
next operands
143
INCREMENT: "ptr"
next
148
FUNCTION DECL
name: malloc
next operands
144
CALL EXPRESSION
next operands
149
ARGUMENT: 10
next
150
FUNCTION DECL
name: free
next
151
ARGUMENT: ptr
next
RETURN 0
145

FIG. 16A

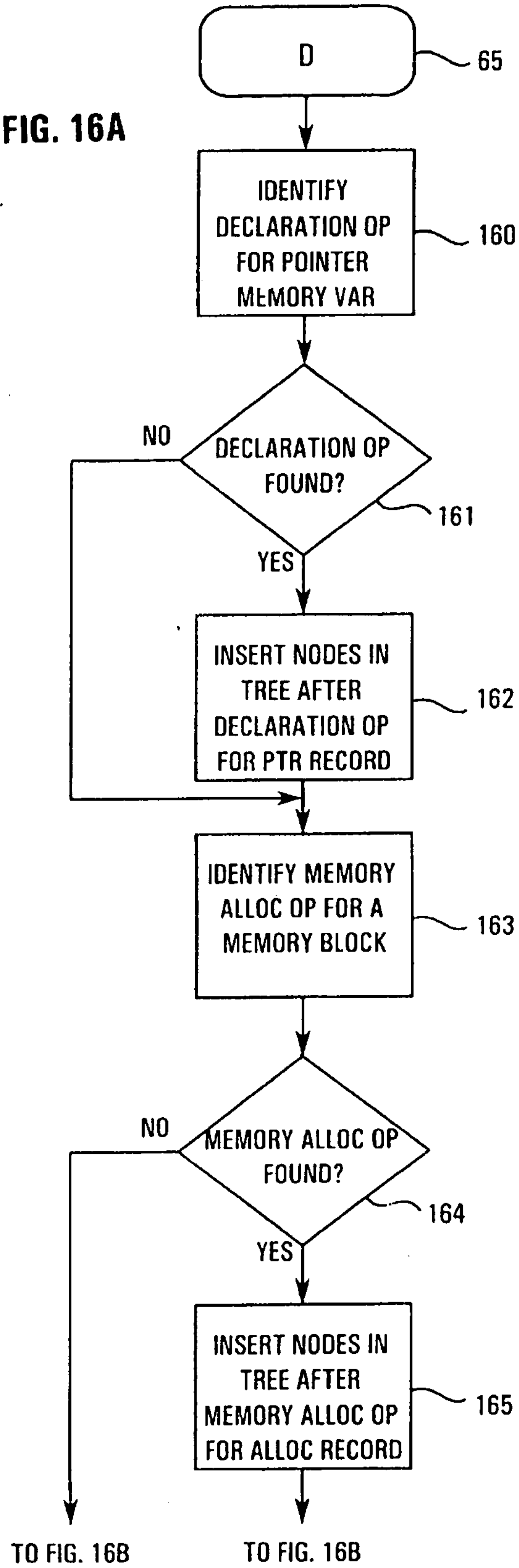

D
65
IDENTIFY DECLARATION OP FOR POINTER MEMORY VAR
160
DECLARATION OP FOUND?
161
NO
YES
INSERT NODES IN TREE AFTER DECLARATION OP FOR PTR RECORD
162
IDENTIFY MEMORY ALLOC OP FOR A MEMORY BLOCK
163
MEMORY ALLOC OP FOUND?
164
NO
YES
INSERT NODES IN TREE AFTER MEMORY ALLOC OP FOR ALLOC RECORD
165
TO FIG. 16B
TO FIG. 16B

FIG. 16B

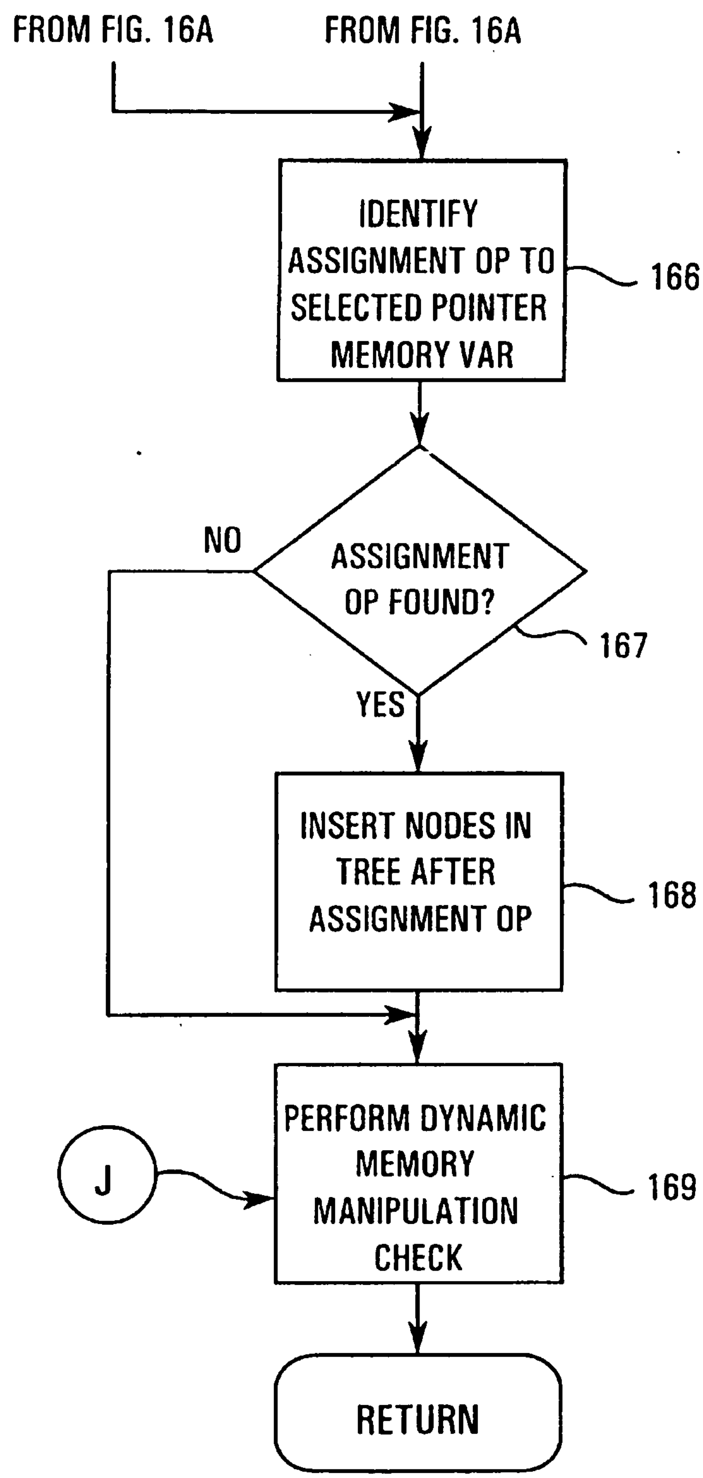
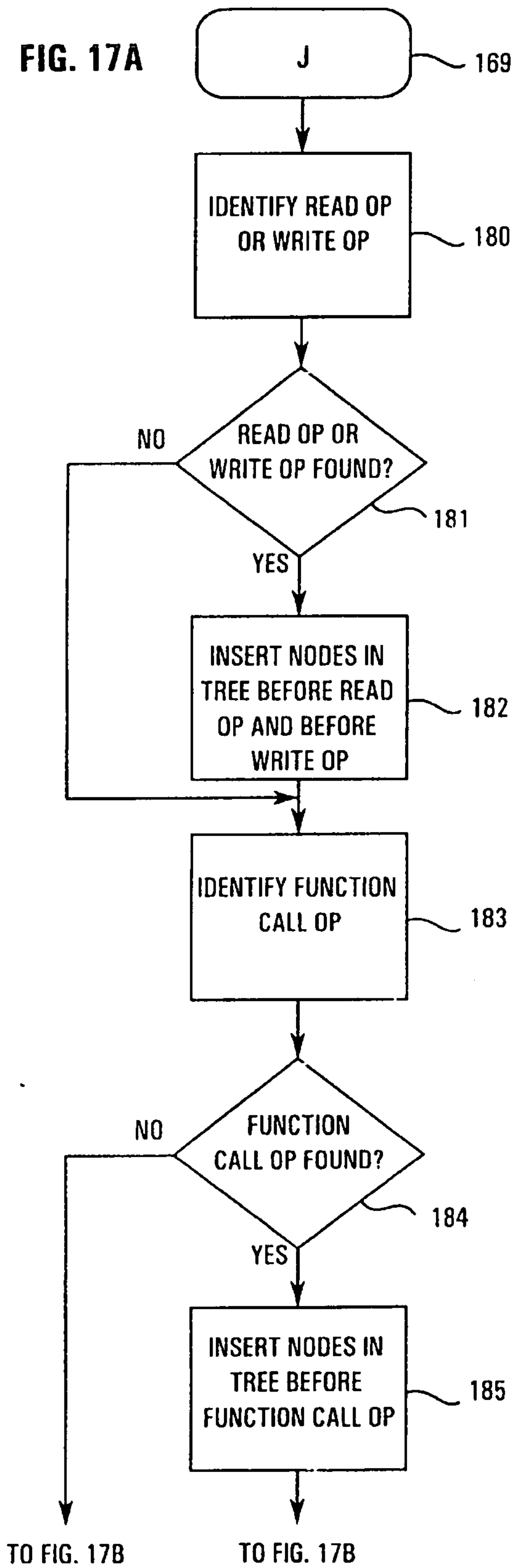

FROM FIG. 16A
FROM FIG. 16A
IDENTIFY ASSIGNMENT OP TO SELECTED POINTER MEMORY VAR
166
ASSIGNMENT OP FOUND?
167
NO
YES
INSERT NODES IN TREE AFTER ASSIGNMENT OP
168
J
PERFORM DYNAMIC MEMORY MANIPULATION CHECK
169
RETURN

FIG. 17A


J
169
IDENTIFY READ OP OR WRITE OP
180
READ OP OR WRITE OP FOUND?
181
NO
YES
INSERT NODES IN TREE BEFORE READ OP AND BEFORE WRITE OP
182
IDENTIFY FUNCTION CALL OP
183
FUNCTION CALL OP FOUND?
184
NO
YES
INSERT NODES IN TREE BEFORE FUNCTION CALL OP
185
TO FIG. 17B
TO FIG. 17B

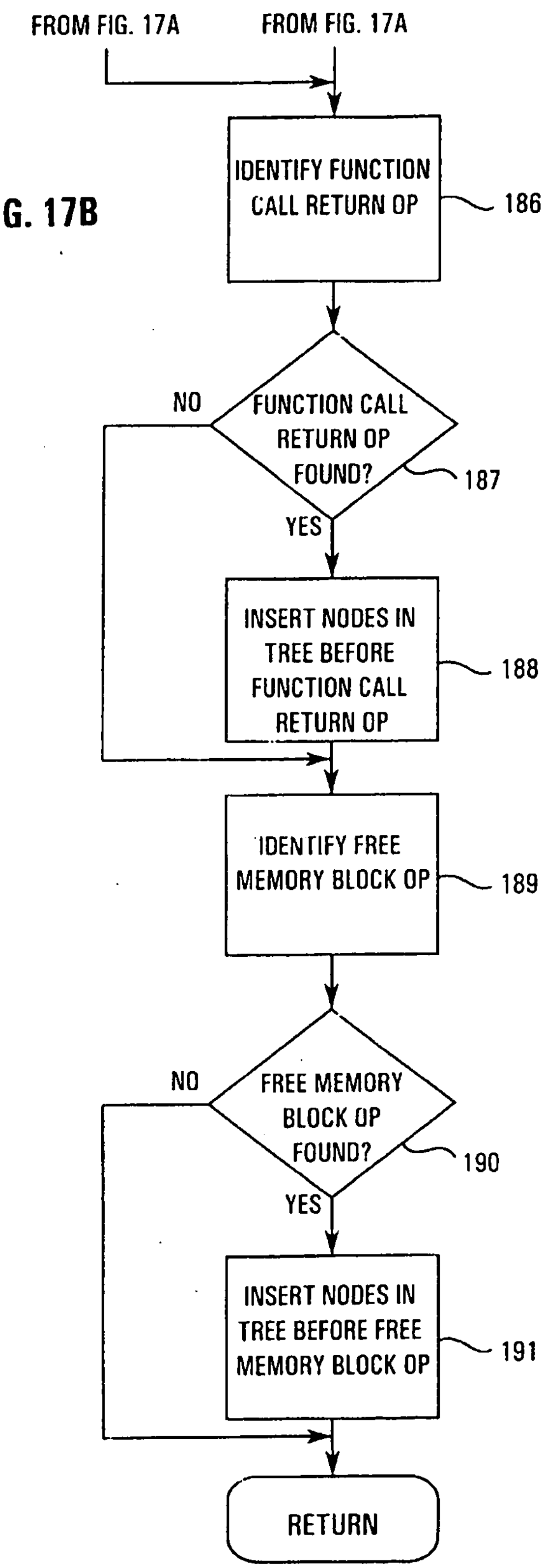

FROM FIG. 17A
FROM FIG. 17A
IDENTIFY FUNCTION CALL RETURN OP
186
FUNCTION CALL RETURN OP FOUND?
187
NO
YES
INSERT NODES IN TREE BEFORE FUNCTION CALL RETURN OP
188
IDENTIFY FREE MEMORY BLOCK OP
189
FREE MEMORY BLOCK OP FOUND?
190
NO
YES
INSERT NODES IN TREE BEFORE FREE MEMORY BLOCK OP
191
RETURN

FIG. 17B

FIG. 18

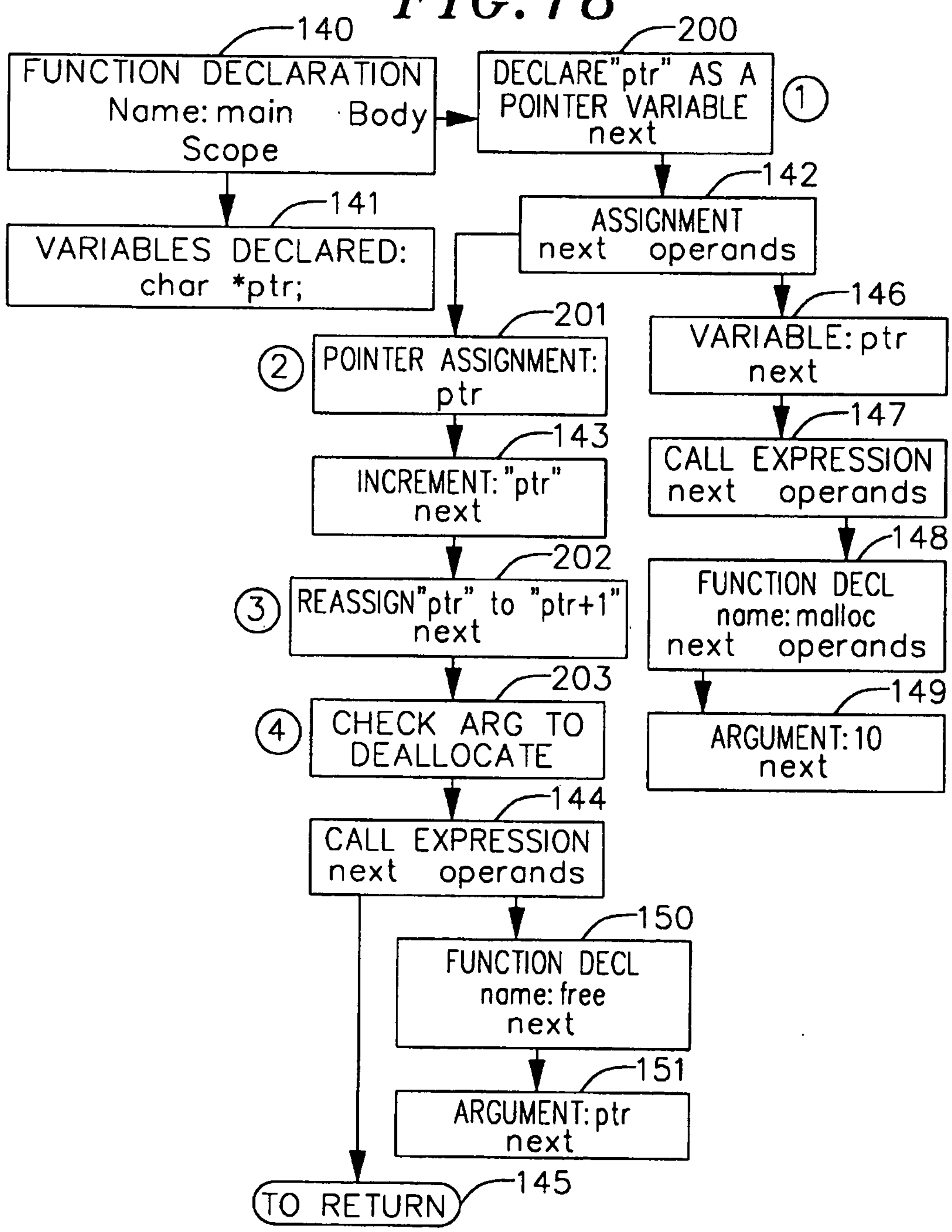

140
FUNCTION DECLARATION
Name: main Body
Scope
141
VARIABLES DECLARED:
char *ptr;
200
DECLARE "ptr" AS A POINTER VARIABLE
next
1
142
ASSIGNMENT
next operands
201
2
POINTER ASSIGNMENT:
ptr
143
INCREMENT: "ptr"
next
202
3
REASSIGN "ptr" to "ptr+1"
next
203
4
CHECK ARG TO DEALLOCATE
144
CALL EXPRESSION
next operands
146
VARIABLE: ptr
next
147
CALL EXPRESSION
next operands
148
FUNCTION DECL
name: malloc
next operands
149
ARGUMENT: 10
next
150
FUNCTION DECL
name: free
next
151
ARGUMENT: ptr
next
145
TO RETURN

FIG. 20

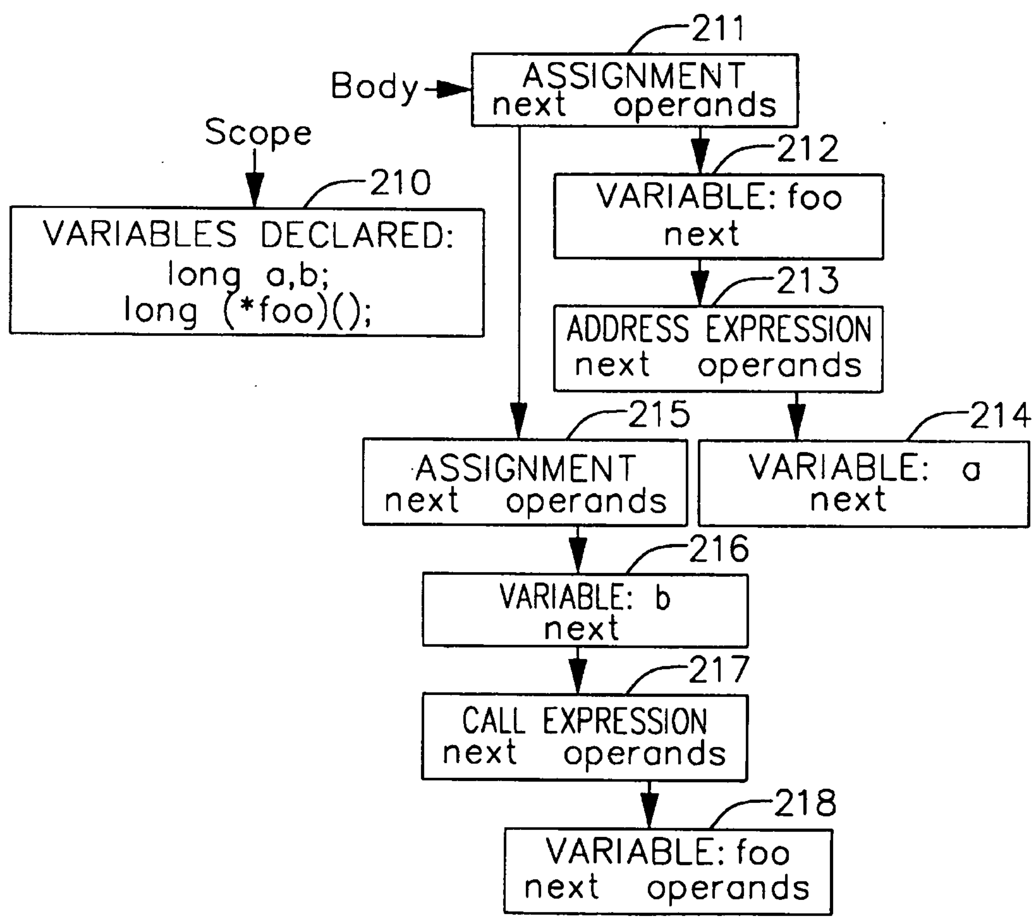

211
Body
ASSIGNMENT
next operands
Scope
212
210
VARIABLE: foo
next
VARIABLES DECLARED:
long a,b;
long (*foo)();
213
ADDRESS EXPRESSION
next operands
215
214
ASSIGNMENT
next operands
VARIABLE: a
next
216
VARIABLE: b
next
217
CALL EXPRESSION
next operands
218
VARIABLE: foo
next operands

FIG. 21A

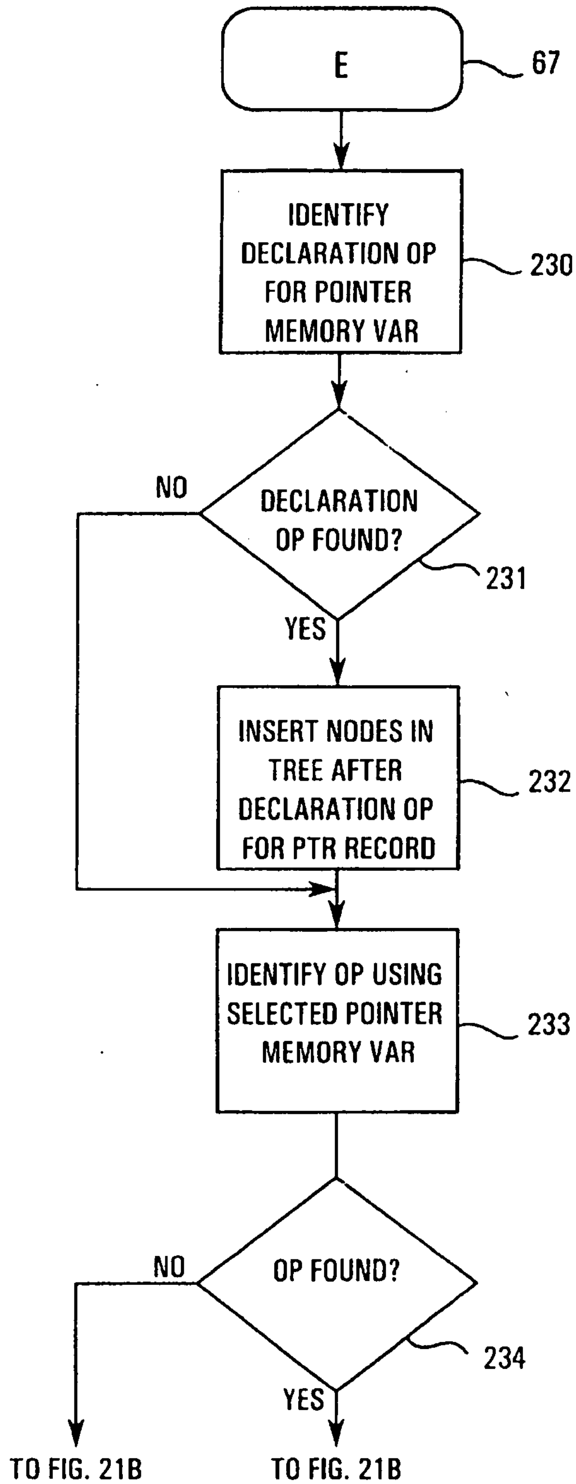

E
67
IDENTIFY DECLARATION OP FOR POINTER MEMORY VAR
230
DECLARATION OP FOUND?
231
NO
YES
INSERT NODES IN TREE AFTER DECLARATION OP FOR PTR RECORD
232
IDENTIFY OP USING SELECTED POINTER MEMORY VAR
233
OP FOUND?
234
NO
YES
TO FIG. 21B
TO FIG. 21B

FIG. 21B

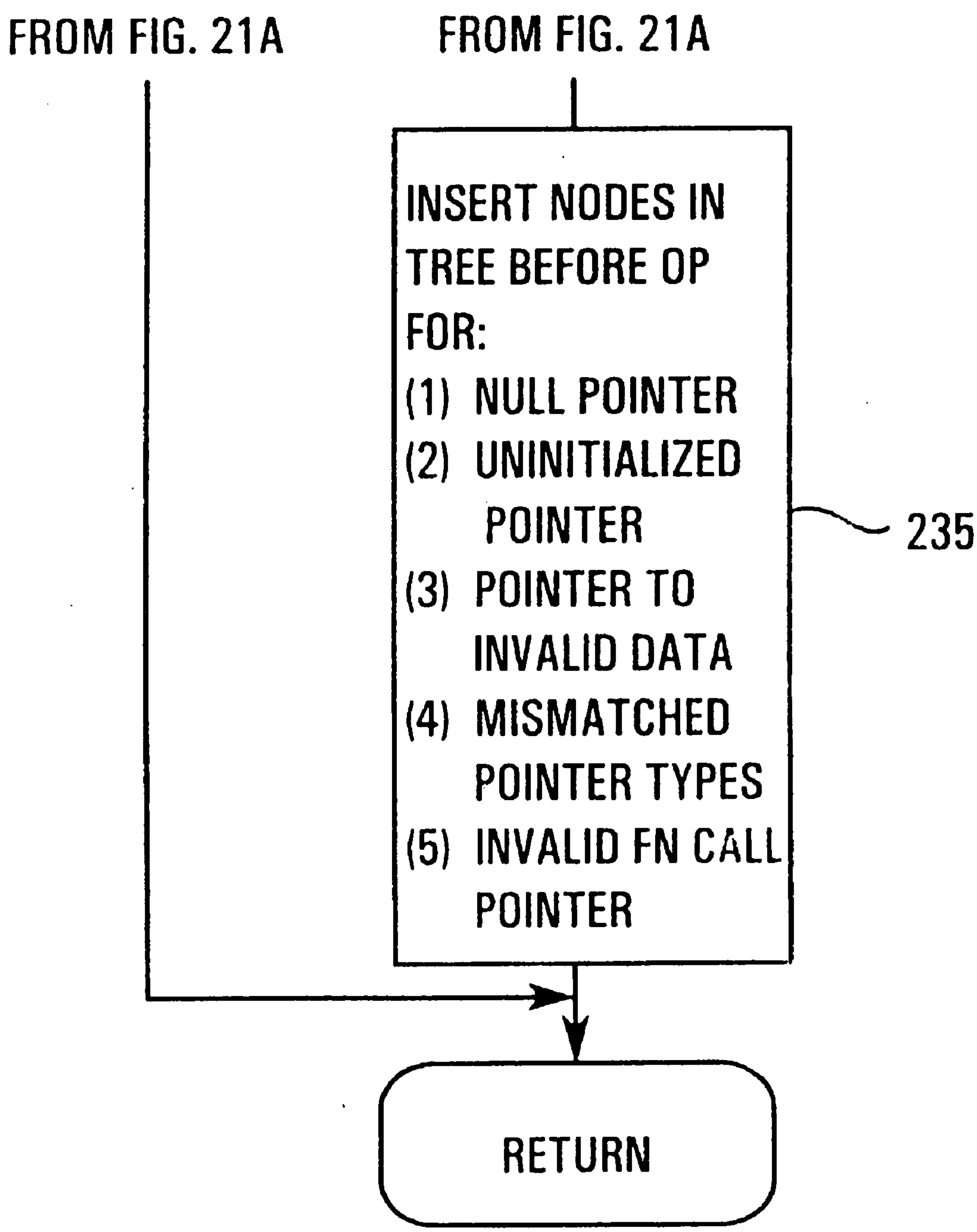

FROM FIG. 21A
FROM FIG. 21A
INSERT NODES IN TREE BEFORE OP FOR:
(1) NULL POINTER
(2) UNINITIALIZED POINTER
(3) POINTER TO INVALID DATA
(4) MISMATCHED POINTER TYPES
(5) INVALID FN CALL POINTER
235
RETURN

FIG. 22

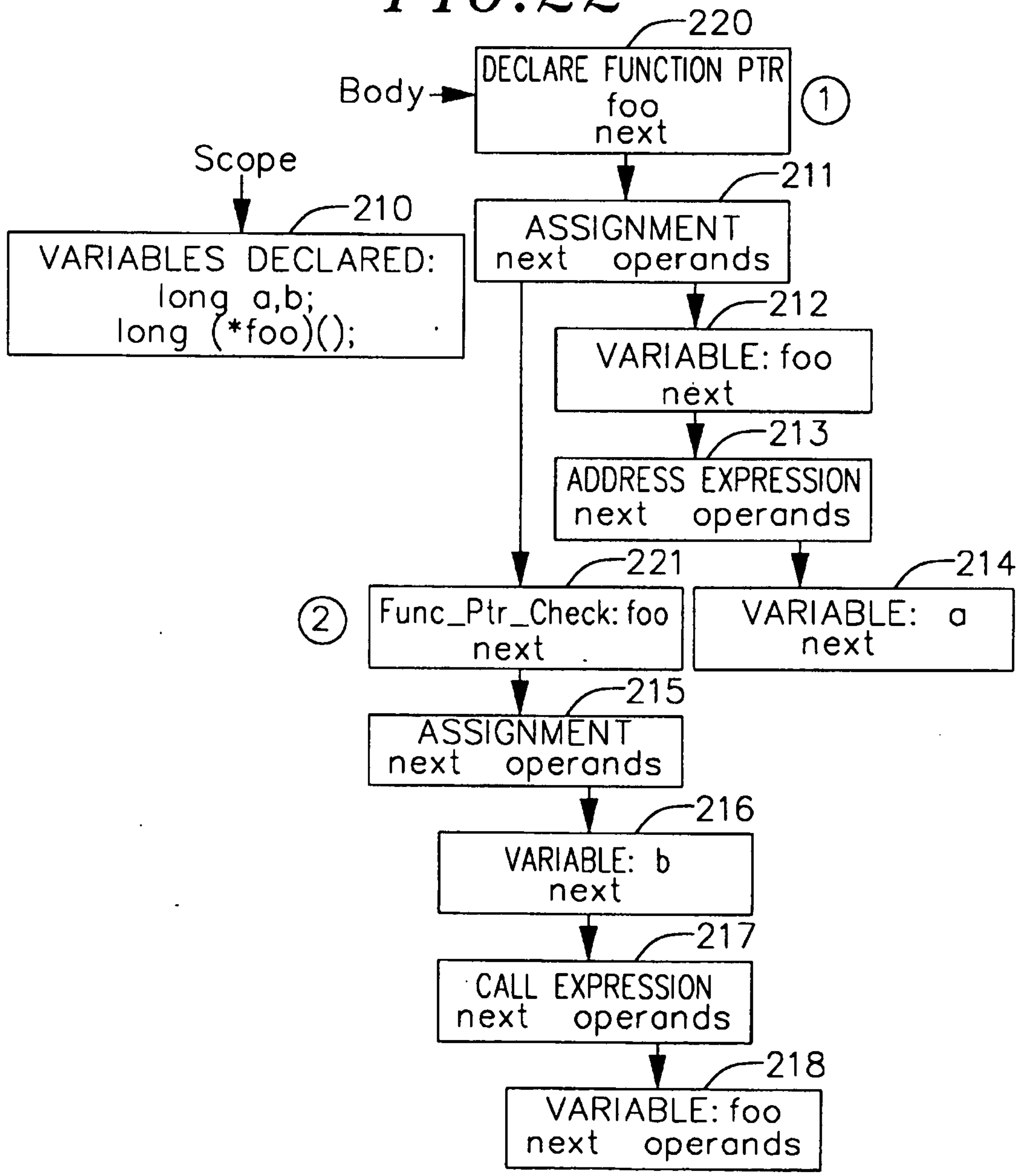

220
Body
DECLARE FUNCTION PTR
foo
next
1
Scope
211
210
VARIABLES DECLARED:
long a,b;
long (*foo)();
ASSIGNMENT
next operands
212
VARIABLE: foo
next
213
ADDRESS EXPRESSION
next operands
221
214
2
Func_Ptr_Check: foo
next
VARIABLE: a
next
215
ASSIGNMENT
next operands
216
VARIABLE: b
next
217
CALL EXPRESSION
next operands
218
VARIABLE: foo
next operands

FIG.23

```
void foo()
{
  char *ptr=malloc(10);
  return;
}
```

FIG.27A

```
ptr=malloc(10);
```

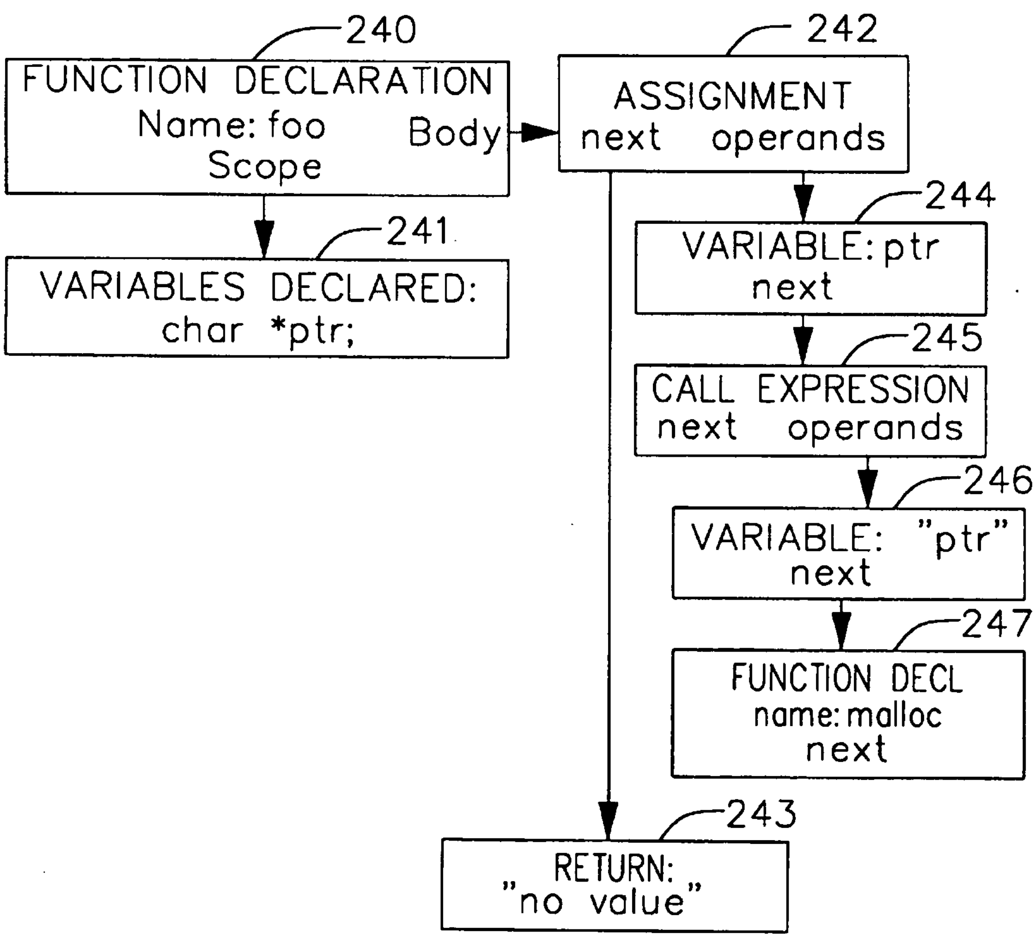
FIG.27B

```
char * malloc(size_t size)
{
 char *a;
 if (size <=0)
  iic_error("Bad malloc size");
 a=malloc(size);
 if (a)
  iic_alloc(a, size);
 else
  iic_error("malloc failed");
 return a;
}
```

FIG.31

```
void foo()
{
 char b, c, *ptr;
 c=getchar();
 if (c < '0' | | c > '9') {
  *ptr=0;
 }
 b=c - '0';
}
```

FIG. 24


240
FUNCTION DECLARATION
Name: foo
Body
Scope
241
VARIABLES DECLARED:
char *ptr;
242
ASSIGNMENT
next operands
244
VARIABLE: ptr
next
245
CALL EXPRESSION
next operands
246
VARIABLE: "ptr"
next
247
FUNCTION DECL
name: malloc
next
243
RETURN:
"no value"

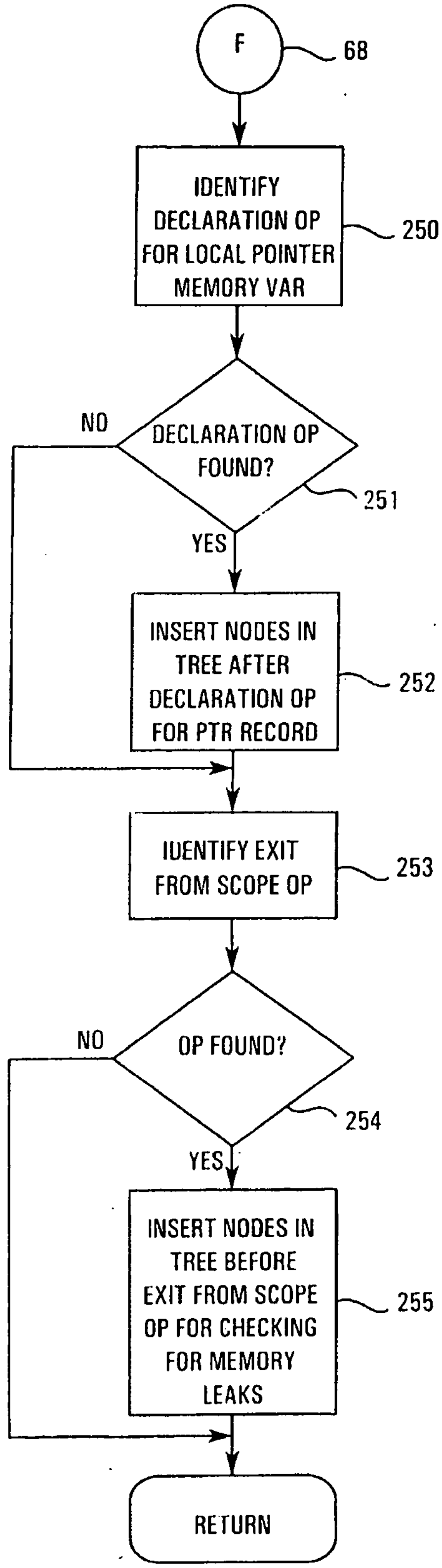

F
68
IDENTIFY DECLARATION OP FOR LOCAL POINTER MEMORY VAR
250
DECLARATION OP FOUND?
251
NO
YES
INSERT NODES IN TREE AFTER DECLARATION OP FOR PTR RECORD
252
IDENTIFY EXIT FROM SCOPE OP
253
OP FOUND?
254
NO
YES
INSERT NODES IN TREE BEFORE EXIT FROM SCOPE OP FOR CHECKING FOR MEMORY LEAKS
255
RETURN

FIG. 25

FIG.26

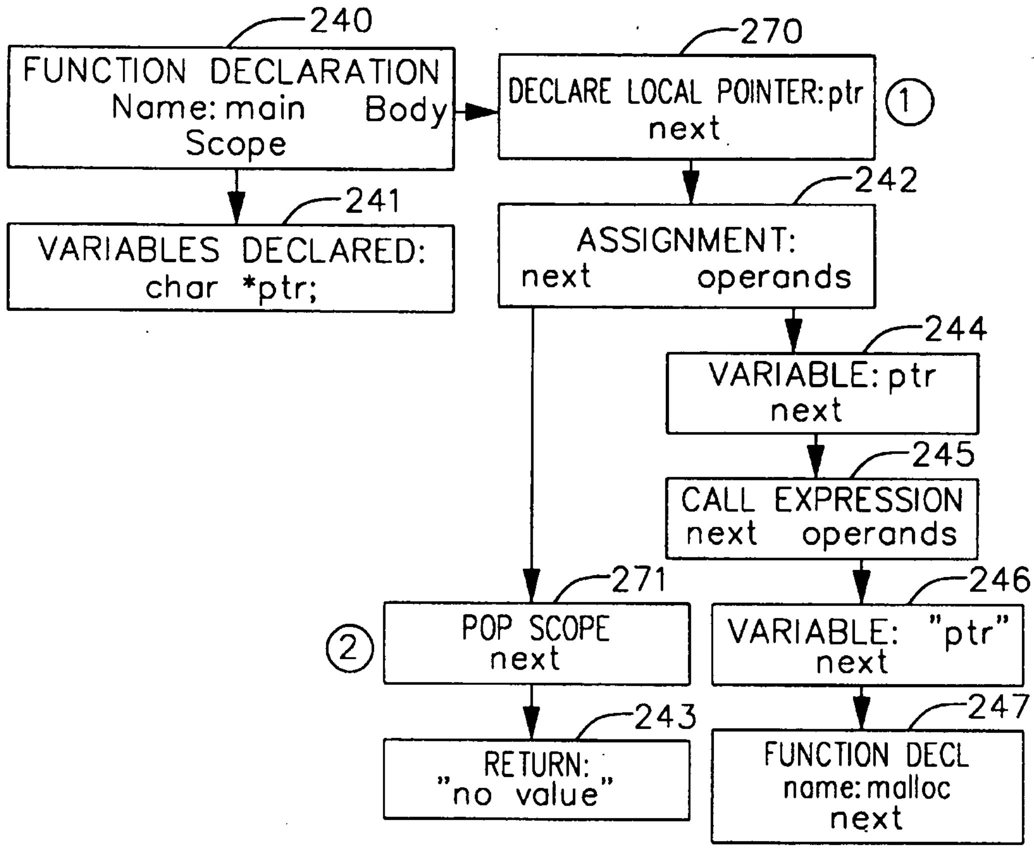

240
FUNCTION DECLARATION
Name: main Body
Scope
241
VARIABLES DECLARED:
char *ptr;
270
DECLARE LOCAL POINTER: ptr
next
1
242
ASSIGNMENT:
next operands
244
VARIABLE: ptr
next
245
CALL EXPRESSION
next operands
271
2
POP SCOPE
next
246
VARIABLE: "ptr"
next
243
RETURN:
"no value"
247
FUNCTION DECL
name: malloc
next

FIG.28

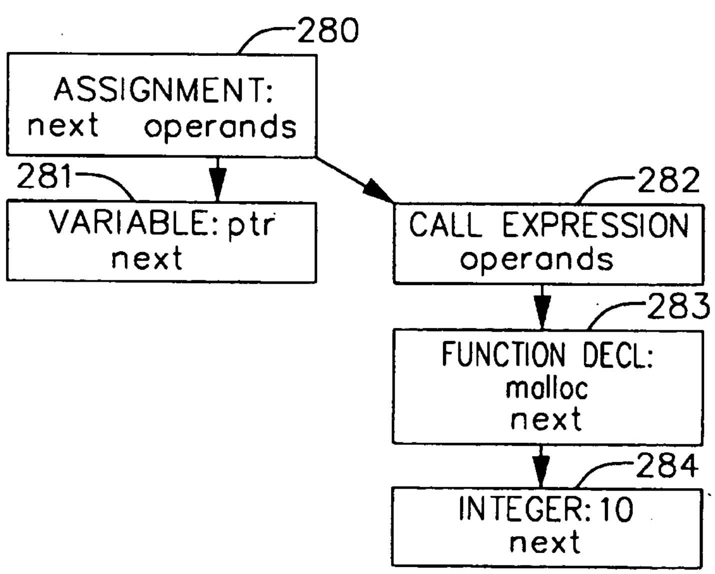
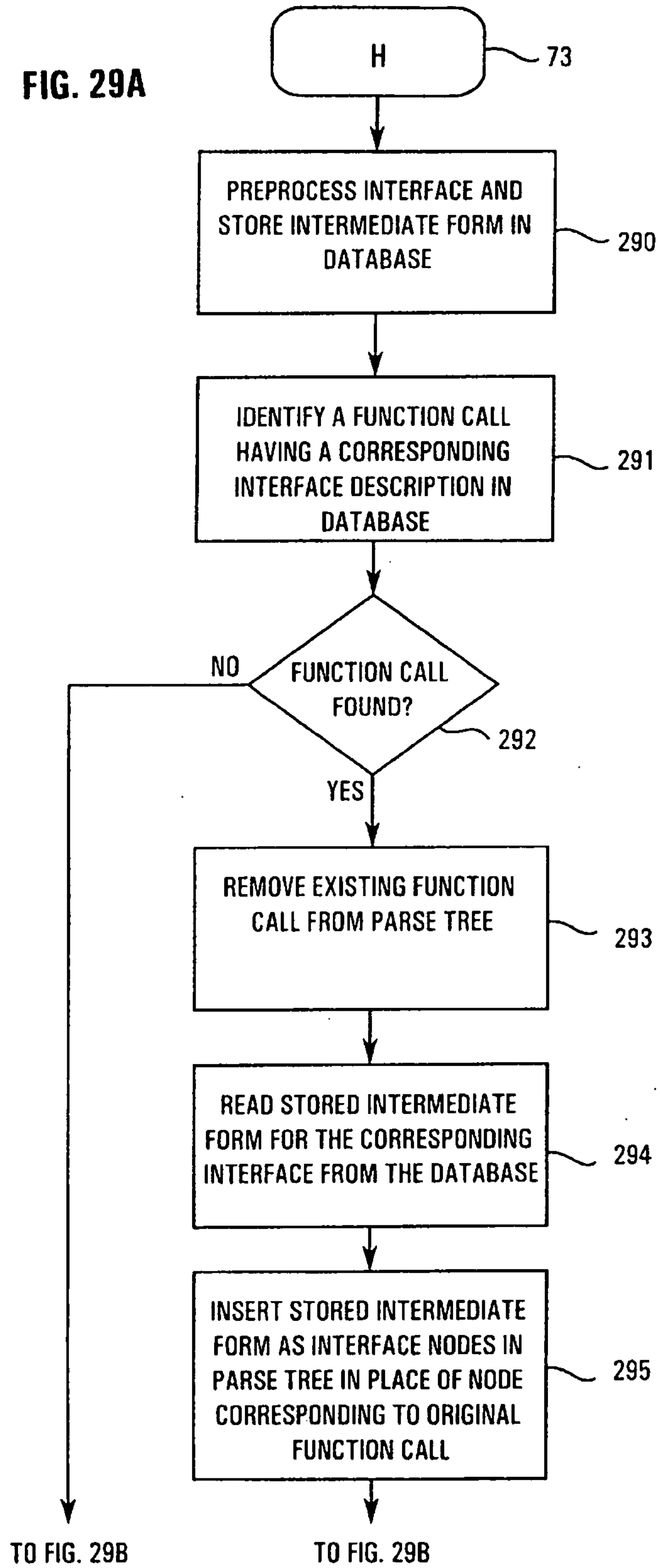

280
ASSIGNMENT:
next operands
281
VARIABLE: ptr
next
282
CALL EXPRESSION
operands
283
FUNCTION DECL:
malloc
next
284
INTEGER: 10
next

FIG. 29A



H
73
PREPROCESS INTERFACE AND STORE INTERMEDIATE FORM IN DATABASE
290
IDENTIFY A FUNCTION CALL HAVING A CORRESPONDING INTERFACE DESCRIPTION IN DATABASE
291
FUNCTION CALL FOUND?
292
NO
YES
REMOVE EXISTING FUNCTION CALL FROM PARSE TREE
293
READ STORED INTERMEDIATE FORM FOR THE CORRESPONDING INTERFACE FROM THE DATABASE
294
INSERT STORED INTERMEDIATE FORM AS INTERFACE NODES IN PARSE TREE IN PLACE OF NODE CORRESPONDING TO ORIGINAL FUNCTION CALL
295
TO FIG. 29B
TO FIG. 29B

FIG. 29B

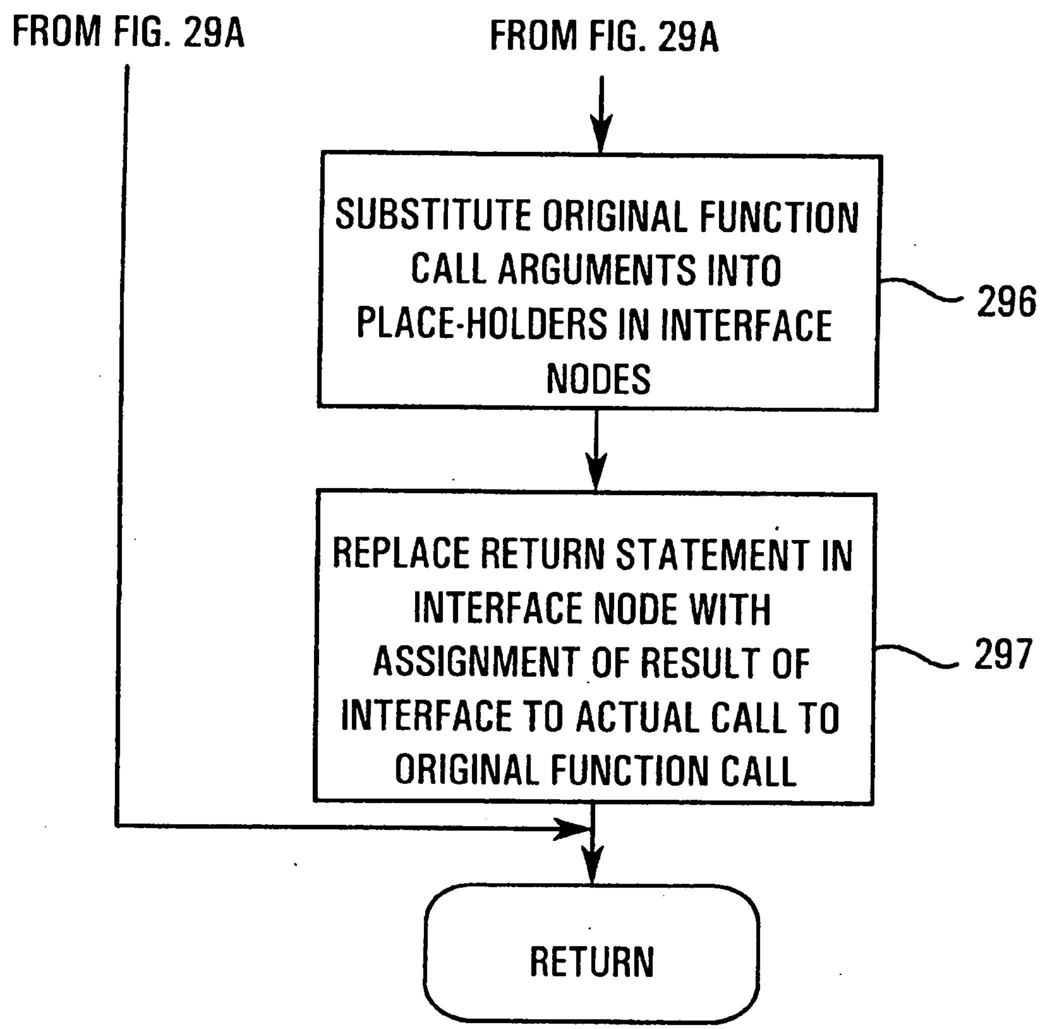
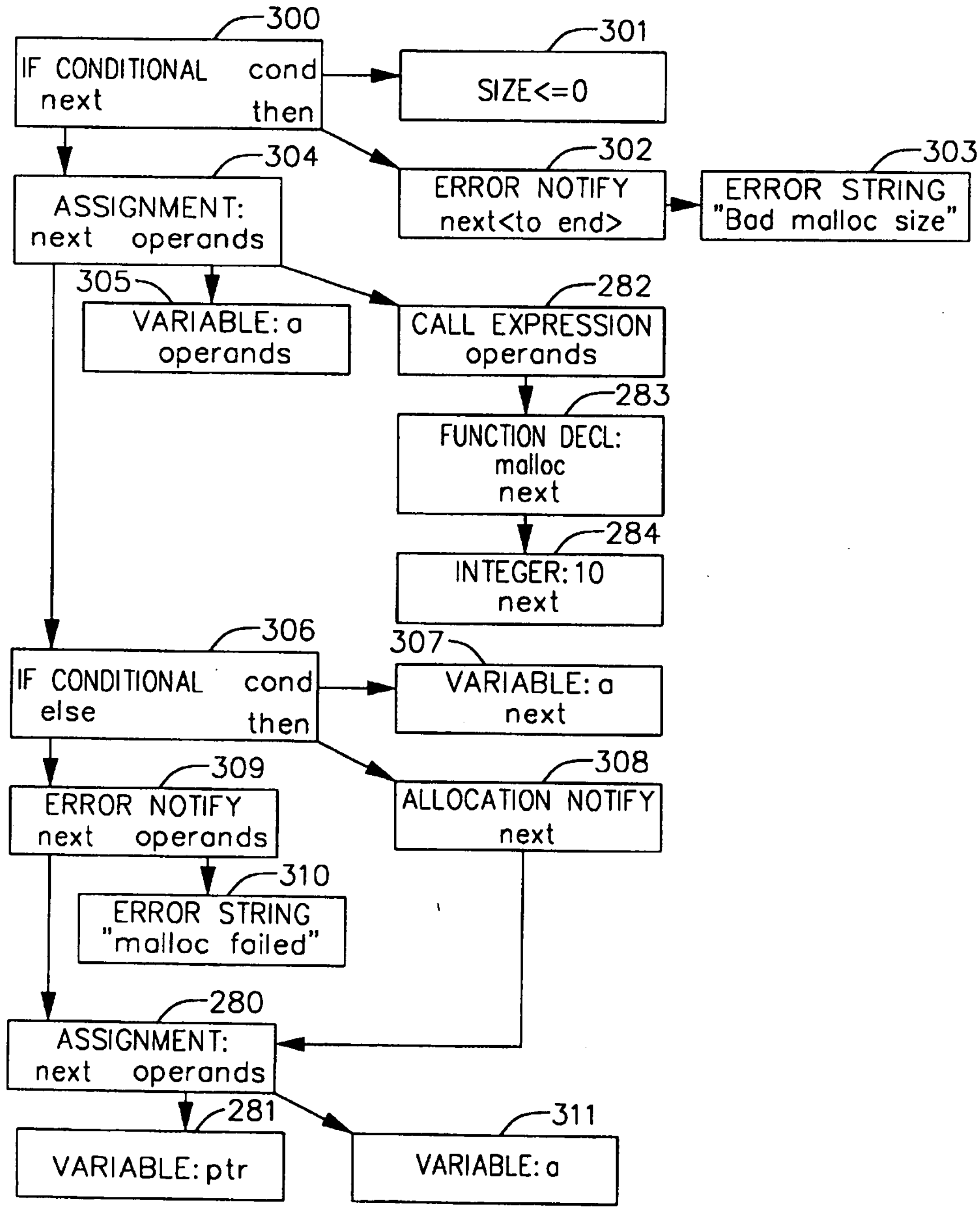
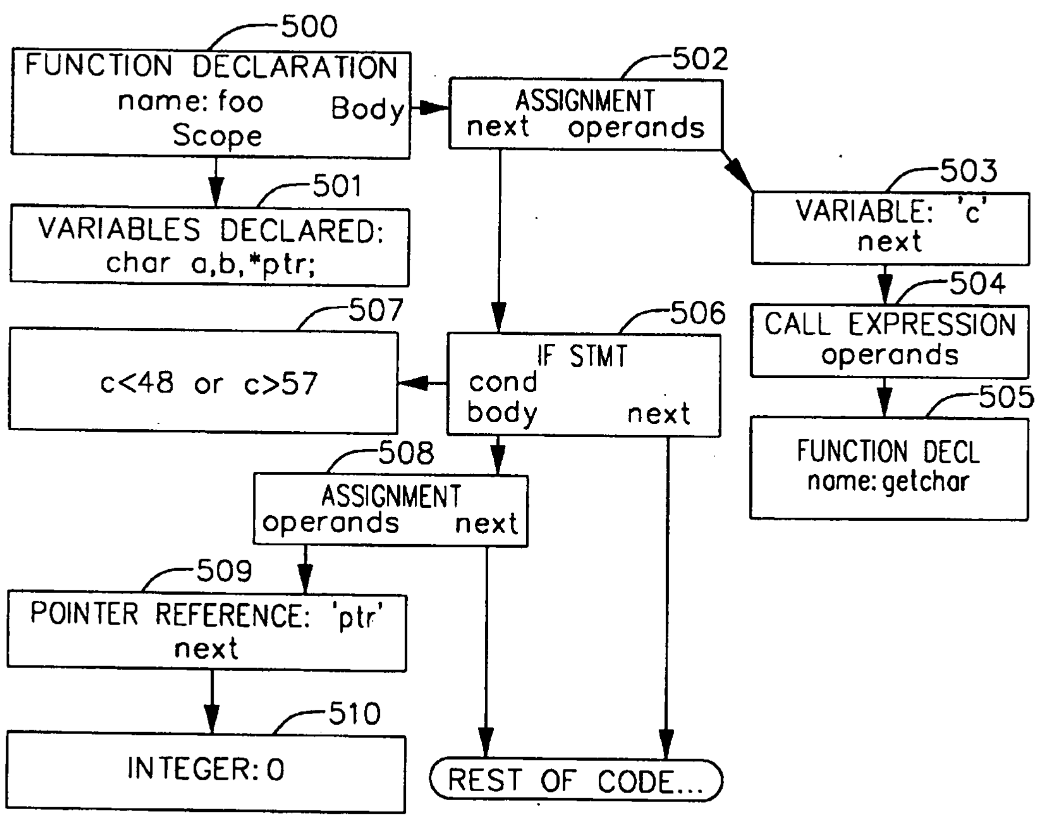

FROM FIG. 29A
FROM FIG. 29A
SUBSTITUTE ORIGINAL FUNCTION CALL ARGUMENTS INTO PLACE-HOLDERS IN INTERFACE NODES
296
REPLACE RETURN STATEMENT IN INTERFACE NODE WITH ASSIGNMENT OF RESULT OF INTERFACE TO ACTUAL CALL TO ORIGINAL FUNCTION CALL
297
RETURN

FIG.30



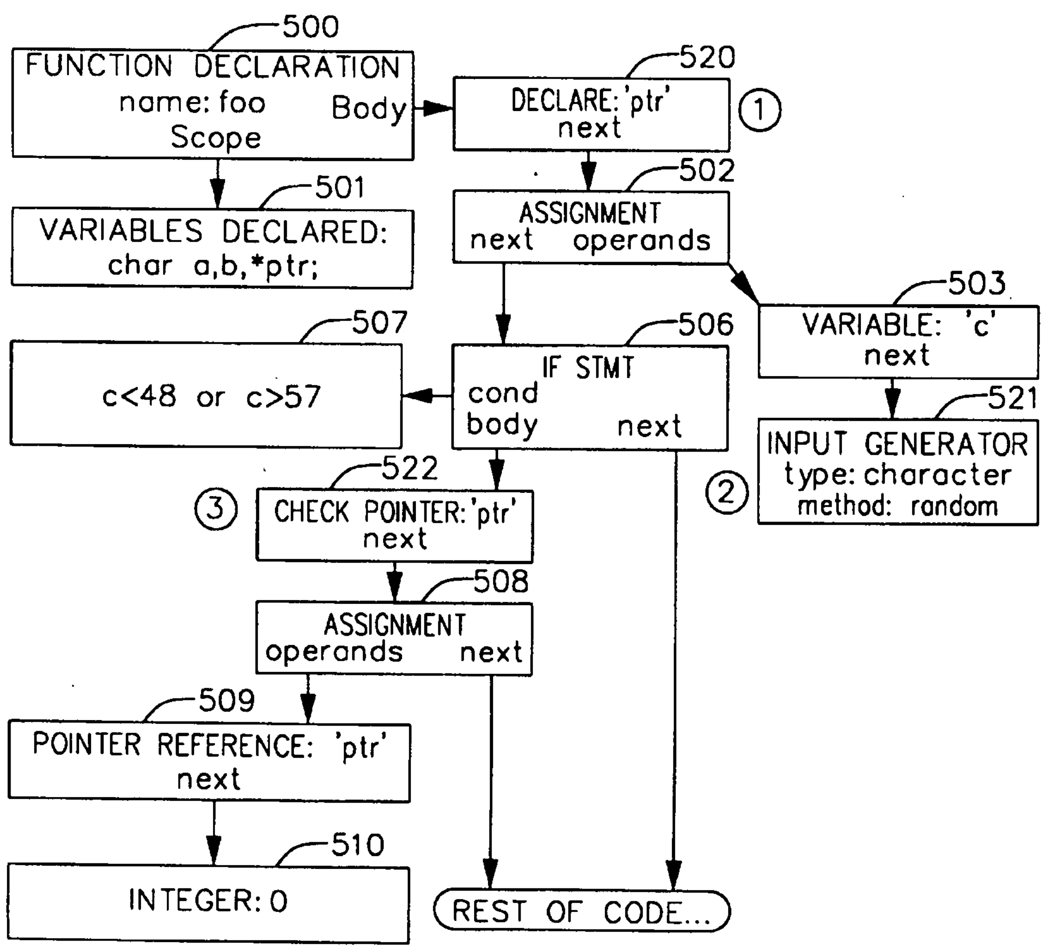
300
IF CONDITIONAL cond
next then
301
SIZE<=0
302
ERROR NOTIFY
next<to end>
303
ERROR STRING
"Bad malloc size"
304
ASSIGNMENT:
next operands
305
VARIABLE: a
operands
282
CALL EXPRESSION
operands
283
FUNCTION DECL:
malloc
next
284
INTEGER: 10
next
306
IF CONDITIONAL cond
else then
307
VARIABLE: a
next
309
ERROR NOTIFY
next operands
308
ALLOCATION NOTIFY
next
310
ERROR STRING
"malloc failed"
280
ASSIGNMENT:
next operands
281
VARIABLE: ptr
311
VARIABLE: a

FIG. 32



500
FUNCTION DECLARATION
name: foo
Scope
Body
501
VARIABLES DECLARED:
char a,b,*ptr;
502
ASSIGNMENT
next operands
503
VARIABLE: 'c'
next
504
CALL EXPRESSION
operands
505
FUNCTION DECL
name: getchar
506
IF STMT
cond
body
next
507
c<48 or c>57
508
ASSIGNMENT
operands next
509
POINTER REFERENCE: 'ptr'
next
510
INTEGER: 0
REST OF CODE...

FIG. 33


500
FUNCTION DECLARATION
name: foo Body
Scope
501
VARIABLES DECLARED:
char a,b,*ptr;
520
DECLARE: 'ptr'
next
1
502
ASSIGNMENT
next operands
503
VARIABLE: 'c'
next
521
INPUT GENERATOR
type: character
method: random
2
507
c<48 or c>57
506
IF STMT
cond
body next
522
3
CHECK POINTER: 'ptr'
next
508
ASSIGNMENT
operands next
509
POINTER REFERENCE: 'ptr'
next
510
INTEGER: 0
REST OF CODE...

METHOD USING A COMPUTER FOR AUTOMATICALLY INSTRUMENTING A COMPUTER PROGRAM FOR DYNAMIC DEBUGGING

CROSS-REFERENCE TO RELATED APPLICATION

This is a continuation of application Ser. No. 08/435,759 filed May 9, 1995 now U.S. Pat. No. 5,581,696.

Reference to Microfiche Appendix and Printed Appendices

A microfiche appendix is part of the specification which includes 17 microfiche and 1585 frames.

In addition, two printed documents are part of the specification and are included as 28-page Appendix A and 17-page Appendix B. Two computer source code listings are also part of this specification and are included as 14-page Appendix C and 3-page Appendix D.

FIELD OF THE INVENTION

The present invention relates in general to automatic instrumentation methods for computer programs and in particular to automatic instrumentation methods for debugging of a computer program using a compiler parse tree.

BACKGROUND OF THE INVENTION

Typically, computer programs are developed in a cycle of writing, compiling, executing, debugging and rewriting computer code until a satisfactory program is attained. Two types of debugging can be performed: static debugging whereby the source code comprising the computer program is analyzed and corrected for errors prior to program execution, and dynamic debugging whereby runtime errors are detected by observing the behavior of the program during execution.

A computer program can be dynamically debugged by employing a separate program or device to observe the behavior of the target computer program by monitoring memory locations. A computer program can also be dynamically debugged internally by introducing debug statements or routines into the program and observing the results during program execution. These statements can be manually introduced into the source code during the writing stage of program development. They can also be automatically introduced by a separate program at some stage in the development cycle prior to execution. The automatic introduction of debug statements or routines is known as instrumentation.

Instrumentation can be used to perform tasks useful to debugging and analyzing a computer program. These include: analyzing code coverage to determine how often each program statement is executed and how long it takes to run; analyzing variables to determine what values are taken on and how often different parts of memory are accessed; analyzing program characteristics, such as memory usage and which functions are called using which parameters; and analyzing the correct use of program code by checking various assertions that ensure that what the program is doing actually makes sense. In addition to the tasks listed above, instrumentation can be used to automatically generate test cases for dynamically testing the program. Test case data for program inputs can be generated automatically by the instrumentation which then links to a test harness program to repeatedly execute the program with different inputs.

Instrumentation can be automatically built into a computer program in a number of ways. First, instrumentation can be introduced before compilation by manipulating the source code and introducing instrumentation routines at appropriate locations. A problem with this approach is that it is slow and inefficient for large or highly complex programs.

Instrumentation can also be automatically introduced after compilation but before link editing by analyzing the relocatable object code output by the compiler. A problem with this approach is that the broader context of the target program is lost to the earlier stages of compilation. Consequently, the introduction of instrumentation must be limited to an analysis of memory locations and address pointers.

Finally, instrumentation can be automatically introduced after link editing by manipulating the executable program. This approach suffers from the same problems as with relocatable object code.

A further problem with these approaches is that the automatic introduction of instrumentation constitutes an extra stage in the program development cycle. Consequently, there is a need for a method of automatically instrumenting a computer program for dynamic debugging as an integral part of the program development cycle and without introducing an extra stage.

SUMMARY OF THE INVENTION

The present invention overcomes the above problems and pertains to a method for automatically instrumenting a computer program for dynamic debugging. More specifically, such a computer program constitutes source code written in a programming language for executing instructions on a computer. The programming language has a grammar comprising operations having an operator and at least one operand and a set of rules for relating each such operator to its respective operand(s). The method consists of the steps of providing the source code as a sequence of statements in a storage device to the computer. Each of the statements are separated into tokens representing either an operator or at least one operand.

A parse tree is built according to the set of rules using the set of tokens whereby the parse tree is a directed acyclic graph and constitutes a plurality of nodes connected by paths organized into a hierarchy of parent nodes representing operators connected to children nodes representing operands of the operators. The parse tree contains embedded error detection statements for communicating information to a runtime error-checking facility which can test for and indicate error conditions as they occur. The parse tree is instrumented to create an instrumented parse tree for indicating that an error condition occurred in the computer program during execution. Object code is generated from the instrumented parse tree and stored in a secondary storage device for later execution using an error-checking engine that indicates error conditions present in the computer program.

BRIEF DESCRIPTION OF THE DRAWINGS

The above and further advantages of this invention may be better understood by reference to the following detailed

description taken in conjunction with the accompanying drawings in which:

FIG. 1 is a block diagram of a process for creating and debugging a computer program;

FIG. 2 is a schematic diagram of a computer system for performing a method for automatically instrumenting a computer program for dynamic debugging according to the present invention;

FIG. 3 is a software block and schematic diagram for a method for automatically instrumenting a computer program for dynamic debugging;

FIG. 4 is a flow chart of a preferred embodiment of the method according to the present invention;

FIGS. 5A, 5B and 5C are a flow chart of a routine for determining the instrumentation to augment a parse tree;

FIG. 6 is a source code listing of a computer program containing an uninitialized read of a memory variable;

FIG. 7 is a diagram illustrating a parse tree representation of the source code listing in FIG. 6;

FIGS. 8A and 8B are a flow chart for a routine for detecting an uninitialized read of a program variable error condition;

FIG. 9 is a diagram illustrating an instrumented parse tree representation of the source code listing shown in FIG. 6;

FIG. 10 is a source code listing of a computer program containing a write operation to an invalid memory address;

FIG. 11 is a diagram illustrating a parse tree representation of the source code listing in FIG. 10;

FIG. 12 is a flow chart for a routine for detecting a write operation to an invalid memory address for a complex memory variable error condition;

FIG. 13 is a diagram illustrating an instrumented parse tree representation of the source code listing shown in FIG. 10;

FIG. 14 is a source code listing of a computer program containing a dynamic memory manipulation error using a pointer memory variable;

FIG. 15 is a diagram illustrating a parse tree representation of the source code listing in FIG. 14;

FIGS. 16A and 16B are a flow chart of a routine for detecting a dynamic memory manipulation error using a pointer memory variable error condition;

FIGS. 17A and 17B are a flow chart of a routine for performing a dynamic memory manipulation check;

FIG. 18 is a diagram illustrating an instrumented parse tree representation of the source code listing shown in FIG. 14;

FIG. 19 is a source code listing of a program segment containing an inappropriate use of a pointer memory variable;

FIG. 20 is a diagram illustrating a parse tree representation of the source code listing shown in FIG. 19;

FIGS. 21A and 21B are a flow chart of a routine for detecting an inappropriate use of a pointer memory variable error condition;

FIG. 22 is a diagram illustrating an instrumented parse tree representation of the source code listing shown in FIG. 19;

FIG. 23 is a source code listing of a computer function containing a memory leak error;

FIG. 24 is a diagram illustrating a parse tree representation of the source code listing shown in FIG. 23;

FIG. 25 is a flow chart of a routine for detecting a memory leak error condition;

FIG. 26 is a diagram illustrating an instrumented parse tree representation of the source code listing shown in FIG. 23;

FIG. 27A is a computer function to be instrumented with an interface and FIG. 27B is the interface routine;

FIG. 28 is a diagram illustrating a parse tree representation of the program segment shown in FIG. 27A;

FIGS. 29A and 29B are a flow chart of a routine for inserting an interface;

FIG. 30 is a diagram illustrating an instrumented parse tree representation of the program segment shown in FIG. 27A;

FIG. 31 is a source code listing of a computer function to be instrumented for automatic test case generation;

FIG. 32 is a diagram illustrating a parse tree representation of the source code listing shown in FIG. 31; and

FIG. 33 is a diagram illustrating an instrumented parse tree representation of the source code listing shown in FIG. 31.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

A block diagram of a process for creating and debugging a computer program is shown in FIG. 1. A source program 10 comprising source code written in a programming language for executing instructions on a computer system is translated into an executable program 13 through a compilation process 11. The source program is translated into an equivalent program that can be run on a target computer system. The compilation process can involve compiling, interpreting or a similar translation of the source program.

The compilation process also reports to the user the presence of static errors 10 in the source program due to errors in lexicography, syntax and semantics. For instance, a string of characters can fail to form a token recognized by the programming language (lexicographic error). Or, a set of tokens may violate a structure rule that the parser is unable to construct into a branch of a parse tree syntactic error). Or, a proper syntactic structure can be semantically incorrect because it fails to have any meaning with respect to the operation involved (semantic error).

After the static errors are resolved, the program is further evaluated during the execution process 14 which detects dynamic errors 15 based on the runtime attributes of program operation. Dynamic errors are difficult to detect since they stem from logical or conceptual errors in the drafting of the source program rather than the concrete static errors resulting from an improper expression of the program. To detect dynamic errors, the program must be instrumented with debug routines during some phase of the compilation process whereby messages indicating the presence of a dynamic error are generated for evaluation by the user.

A schematic diagram of a computer system for performing a method for automatically instrumenting a computer program for dynamic debugging according to the present invention is shown in FIG. 2. A main memory 23 contains a compiler 24 in the form of a computer program for carrying out the steps of compiling and a code instrumenter 28 for automatically instrumenting a computer program. A microprocessor 22 runs the compiler using the source program file 20, which contains the source program 10, and the programming language definition file 21, which contains a grammar comprising operations and a set of rules. The microprocessor runs the compiler and creates an executable program file 25, which contains the instrumented executable program 13 in the form of object code.

During the execution of the compiler **24**, an error may arise due to some problem with the source program. Error messages are presented to the user on a display **26** and collected and stored in an error file **27**.

Source code listings for a computer program for use in one embodiment of the present invention are included in the microfiche appendix. The source code is written in C language. A description of the C language is detailed in B. W. Kernighan & D. M. Ritchie, *The C Programming Language*, Prentice Hall (2d Ed. 1988), the disclosure of which is hereby incorporated by reference.

The computer program of the microfiche appendix is preferably run on a Sun Microsystems SPARCstation **20** workstation running the Unix operating system. The source code listings are compiled using the instructions contained in Appendix D, the disclosure of which is hereby incorporated by reference. The resulting program is executed. Preferably, the workstation is equipped with 64 megabytes of random access memory and 4 gigabytes of secondary storage space.

A software block and schematic diagram for a method for automatically instrumenting a computer program for dynamic debugging according to the present invention is shown in FIG. **3**. One embodiment of the present invention is described in "Compiler Intermediate Code Insertion for Automatic Debugging and Test Case Generation," which is attached as Appendix A, the subject matter of which is hereby incorporated by reference as if set forth in full. The method uses a computer program consisting of five main components coordinated by a main control component **30**. The source code component **31** reads a source code file **32** stored in a secondary storage device and provides it to the computer program.

A lexical analysis component **33** separates the sequence of statements making up the source code in to tokens **34** by scanning the characters comprising each statement and grouping the characters into tokens. Each token has a collective meaning in the context of the grammar defining the programming language that the source program is written in. In most programming languages, key words, operators, identifiers, constants, literal strings, and punctuation symbols (such as parentheses, commas and semicolons) are treated as tokens. The tokens **34** are stored in the main memory.

A parsing and semantic analysis component **35** groups the tokens into grammatical phrases that are used to represent the instructions to be performed by the source program. These grammatical phrases are represented by a parse tree **36**, which is stored in main memory.

The parse tree describes the syntactic structure of the source program. A description of the data structures used for representing a parse tree in one embodiment of the present invention is attached as Appendix C and the subject matter of which is hereby incorporated by reference. It is a hierarchical representation of the instructions making up the program structured as a directed acyclic graph comprising a hierarchy of parent and children nodes interconnected by paths with a root node representing the program's entry point. The blueprint for creating a parse tree is provided by the rules of the programming language grammar. Each path connecting a parent node to a child node represents a relationship between an operator and its operands. A single instruction can comprise several operations and each such operation becomes a node in the parse tree. Operations can be defined recursively whereby an operation constitutes an operand for another operation.

An instrumentation component **37** reads the stored parse tree and augments the parse tree with instrumentation for use in dynamic debugging. The details of the instrumentation component are described in more detail below. It generates an instrumented parse tree **38**. In a preferred embodiment of the present invention, eight categories of instrumentation are used. These include detecting a read operation to an uninitialized memory variable, detecting a read or write operation to an invalid memory address for a complex memory variable, detecting a dynamic memory manipulation error using a pointer memory variable, detecting an inappropriate use of a pointer memory variable, detecting a memory leak error, and detecting a function call argument error. These also include a user definable instrumentation routine known as an interface and an automatic test case generation routine.

For each category, an analysis is performed to determine which check or operation is appropriate and instrumentation is embedded into the parse tree. Some categories require instrumentation to be introduced in several locations in the parse tree. The result is an instrumented parse tree, which is stored in main memory.

A code generation component **39** reads the instrumented parse tree and generates an object code file **40** with the instrumentation incorporated. This component is sometimes divided into an intermediate code generator, a code optimizer, and a code generator. The intermediate code generator transforms the instrumented parse tree into an intermediate representation representing a program for an abstract machine. This representation is useful for computing expressions, handling flow of control constructs and procedure calls. The code optimizer attempts to improve the performance of the intermediate code by decreasing the running time of the executable program. Finally, the code generator creates relocatable machine code or assembly code to be output as the object code file **40**. Memory locations are selected for each variable used and the intermediate instructions are translated into a sequence of machine instructions that perform the same task. These are combined and output as object code.

Throughout the operation of each component shown in FIG. **3**, reference is made to a language file **41** containing the definition of grammar rules for the programming language. Similarly, errors in the source program that are detected are output to the user through error messages **43** and error file **42**.

A preferred embodiment of the compiler **24** is shown in FIG. **4**. A file containing source code comprising the computer program to be instrumented is provided to the compiler (block **51**). The source code is written in a programming language for executing instructions on a computer.

The programming language is defined by a grammar comprising operations having an operator (to identify the operation) and at least one operand (upon which the operation is performed). In addition, the grammar includes a set of rules for relating each of the operations to their respective operands. Preferably, the grammar is a context-free grammar having four components: a set of tokens, known as terminal symbols; a set of nonterminals; a set of productions, where each production consists of a nonterminal, an arrow, and a sequence of tokens and/or nonterminals; and a designation of one of the nonterminals as a start symbol. The productions define the set of operations comprising that grammar. Each production is structured with the nonterminal on its left side, followed by an arrow, followed by a sequence of tokens and/or nonterminals on its right side. A description of a context-free grammar is detailed in H. R. Lewis & C. H.

Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall (1981), the disclosure of which is hereby incorporated by reference.

Each separate source code statement is separated into tokens (block **52**), each token representing a terminal symbol in the grammar. A token can be either an operator or an operand. In addition, source code comments and white space (comprising blanks, tabs and new line characters) are removed during this step.

The set of tokens is used to build a parse tree (block **53**) that represents the structure of the program operations. The parse tree is structured with certain properties. This includes having a root node labeled by a start symbol, each node being labeled by a token or a null value, and each interior node being labeled with a nonterminal. For each nonterminal node, the children of that node correspond to the right-hand side of the production rule for the operation represented by the parent node. In addition to parse tree representations, other intermediate representations for organizing tokens are possible. The same approach presented herein applies to other intermediate representations as well.

Once completed, the parse tree is instrumented (block **54**) to communicate runtime information to the error-checking engine to facilitate automatic detection of dynamic errors in the source program. This step requires a two-phase approach. During the first phase, the source code is analyzed using a flow analysis procedure to determine the type of instrumentation that is appropriate. During the second phase, the parse tree is augmented with additional nodes comprising the operations required to communicate runtime conditions to the error-checking engine which include appropriate checks for runtime or dynamic errors or programmatic anomalies to the error-checking engine in the form of debug output.

The instrumented parse tree is used to generate code for the target program which not only functions as was originally intended, but also contains calls to instrumentation procedures which provide automatic error detection of dynamic program errors as well as an ability to automatically generate test cases. This is accomplished by passing runtime information to the error-checking engine which is linked with the target program when the program executes.

The instrumented parse tree is used to generate object code (block **55**), which is stored in a secondary storage device. The steps of separating source code into tokens (block **52**), building a parse tree (block **53**), and generating object code (block **55**) are described in A. V. Aho et al., *Compilers, Principles, Techniques and Tools*, Addison-Wesley (1986), the disclosure of which is hereby incorporated by reference.

Referring to FIGS. **5**A, **5**B and **5**C, a routine for instrumenting a parse tree according to the present invention is shown. A step-wise procedure is followed to insert each of the seven categories of instrumentation into the parse tree. Thus, nodes are inserted for detecting a read operation to an uninitialized memory variable (block **61**), detecting a write operation to an invalid memory address for a complex memory variable (block **62**), detecting a dynamic memory manipulation error using a pointer memory variable (block **65**), detecting an inappropriate use of a pointer memory variable (block **67**), detecting a memory leak error (block **69**), inserting a user-defined instrumentation routine (interface) (block **73**), and inserting an automatic test case generation routine (block **75**).

The seven categories of instrumentation perform checks or augment the functionality of the original source code. In addition, other information is communicated to the error-checking engine through the instrumented code. This consists of declarations of variables and pointer addresses and their sizes, assignments of pointers, function entry and exit point indicators, and memory allocation indicators.

The first category of dynamic memory error is the use of uninitialized memory variables. This means that a memory variable is declared, but is not yet assigned a value before it is used by some other statement in the program. Referring to FIG. **6**, a source code listing of a computer program containing an uninitialized read of a memory variable is shown. On line **3**, an integer variable "i" is defined. On line **4**, the variable "i" is read. Since variable "i" is uninitialized, a dynamic error occurs at runtime.

Referring to FIG. **7**, a diagram illustrating a parse tree representation of the source code listing in FIG. **6** is shown. The function declaration (lines **1**, **2** and **6**) is represented by function declaration node **80**. The declaration of variables "a" and "i" (line **3**) is represented by variable declarations node **81**. The assignment operation (line **4**) is represented by assignment node **82** which is followed by return node **83** (corresponding to line **5**). The assignment operation (line **4**) has two operands, variables "a" and "i", represented respectively by variable nodes **84** and **85**. To detect the use of the uninitialized memory variable "i" (line **4**), the parse tree shown in FIG. **7** must be instrumented with debugging functionality so that the attempted assignment statement using variable "i" can be automatically detected by the error checking engine when the program is executed.

The overall criteria for inserting such an error check is as follows. If there is a memory variable used in a program expression that is not known to have been assigned a value previously, an error check is inserted into the parse tree to check that variable during execution. In addition, an error check is inserted to let the error-checking engine know that the variable at that particular address in memory is initially uninitialized at the start of execution.

Referring to FIGS. **8**A and **8**B, a flow chart for a routine for detecting an uninitialized read of a memory variable error condition is shown. First, a memory address is retrieved from a program stack frame (block **90**) which represents memory locations of local memory variables. A flow analysis is performed on the source code to identify any read operation to the memory address for which it cannot be statically determined that the variable has been previously initialized (block **91**). If a read operation is found (block **92**), instrumentation nodes are inserted into the parse tree in two locations. First, nodes are inserted after the parse tree node corresponding to the stack frame containing the memory address for the program variable to be checked (block **93**). These nodes are for setting an internal indication to the error-checking engine that the memory variable is uninitialized. Second, instrumentation nodes are inserted into the parse tree before the read operation (block **94**) to indicate to the error-checking engine that the memory variable being read by the read operation is either initialized or uninitialized at that point in program execution. These nodes determine the status of the memory variable by referring to the indication set by the instrumentation nodes for the stack frame.

Next, a flow analysis is performed on the source code to identify a write operation to the memory address for the program variable being checked, since any write operation will cause the memory variable in question to be initialized (block **95**). If a write operation is found (block **96**), instrumentation nodes are inserted into the parse tree after the

nodes corresponding to the write operation for setting an indication used by the error-checking engine to indicate that the memory variable in question is initialized (block **97**).

Referring to FIG. **9**, a diagram illustrating an instrumented parse tree representation of the source code listing shown in FIG. **6** is shown. The instrumentation nodes for the stack frame are inserted as uninitialized node **100**. The instrumentation nodes for the read operation are inserted as check variable read node **101**. The instrumentation nodes for the write operation are inserted as copy initialize node **102**. Uninitialized node **100** indicates to the error-checking engine that the variables "a" and "i" are uninitialized. Check variable read node **101** indicates to the error-checking engine that an actual check for the uninitialized variable "i" should be performed. Finally, copy initialize node **102** indicates to the error-checking engine that the variable "a" is being assigned a value which initializes it by copying a value from some other memory location.

The second category of dynamic error is a write operation to an invalid memory address for a complex memory variable. This is also known as memory corruption which occurs when a program writes to a location in memory that is not valid. For instance, this can happen as a result of writing off of the end of an array. Similarly, it can happen as result of writing to a location in memory that falls outside of the range of memory locations allocated to a complex memory variable, such as a structure.

A complex memory variable comprises a plurality of elements, each of which can be a constant value, a simple memory variable or a complex memory variable. An array comprises a plurality of identical elements, each of which can be constant value, a simple memory variable, or a complex memory variable.

Referring to FIG. **10**, a source code listing of a computer program containing a write operation to an invalid memory address is shown. An array "A" is defined comprising ten integer elements (line **3**). Each of these ten elements are initialized to 0 (line **5**) using a loop beginning at an index value of 1 (lines **4** and **6**). The valid indices for the array "A" are 0 through 9. However, the loop begins with an index "i" equaling 1 that is incremented during each successive iteration until the index "i" equals 10 (line **4**). Thus, in the tenth iteration, the program attempts to set array element A[**10**] to 0. This is invalid since array "A" does not have an index value of 10 and therefore an overwrite dynamic error occurs.

Referring to FIG. **11**, a diagram illustrating a parse tree representation of the source code listing in FIG. **10** is shown. The function declaration (lines **1**, **2** and **8**) is represented by function declaration node **110**. The declaration of index variable "i" and array "A" which has 10 elements (line **3**) is represented by variable declarations node **111**. The loop operation (lines **4** and **6**) is represented by loop node **112** which is followed by return node **113** (corresponding to line **7**). The assignment operation (line **5**) has two operands, an array element "A[i]" and an integer constant 0, represented respectively by nodes **115** and **116**.

To detect an array operation that is attempting to access an invalid memory location, the parse tree shown in FIG. **11** must be instrumented with debugging functionality so that the error can be automatically detected by the error-checking engine when the program is executed. Here, the array operation is an assignment to element A[**10**] on line **5** of the program. Element A[**10**] is out of bounds.

The overall criteria for inserting this type of error check is as follows. For arrays, the array variable and its size must be declared to the error-checking engine. For each write operation to that array, the error-checking engine must check if the index into the array is valid. For complex memory variables, a similar declaration must be made to the error-checking engine; however, the engine must perform a test for whether the memory address being written to falls outside of a valid memory address range defined by the dimension operand used to declare the memory block size for the complex memory variable.

Referring to FIG. **12**, a flow chart for a routine for detecting a write operation to an invalid memory address for a complex memory variable error condition is shown. This involves a more general error check than for an array and is therefore presented initially.

First, a flow analysis is performed on the source code to identify a declaration operation for a complex memory variable (block **120**). Such a variable comprises a plurality of elements, each of which can be a constant value, a simple memory variable or a complex memory variable. A declaration operation for a complex memory variable comprises two components: an identifier operand for identifying the variable and a dimension operand for identifying a memory block size. If a declaration operation is found (block **121**), instrumentation nodes are inserted after the parse tree node corresponding to the declaration operation (block **122**). These nodes are for storing the dimension operand for use by the error-checking engine during execution.

Next, a flow analysis is performed on the source code to identify a write operation using the complex memory variable being checked (block **123**). If a write operation is found (block **124**), instrumentation nodes are inserted into the parse tree before the nodes corresponding to the write operation (block **125**). During execution, the error-checking engine can indicate that the write operation is writing to an invalid memory address falling outside of the memory address range defined by the stored dimension operand.

To check for a write to an invalid array memory location, an additional step is required to those shown in FIG. **12**. It comprises augmenting the last step with inserting instrumentation nodes into the parse tree to further indicate to the error-checking engine that a write operation is being performed on an array element falling outside of the range of valid array indices.

Referring to FIG. **13**, a diagram illustrating an instrumented parse tree representation of the source code listing shown in FIG. **10** is shown. The instrumentation nodes for the declaration operation are inserted as declare node **130**. The instrumentation nodes for the write operation are inserted as check array access node **131**. Declare node **130** indicates to the error-checking engine that the array "A" is declared and contains ten elements. Since the present program is written in C language and array indices begin with 0, the range of valid indices for array "A" are from 0 through 9. The check array access node **131** indicates to the error-checking engine that a write operation is being performed on an element of array "A" and that the value of the index, here index "i", should be checked to determine whether it falls within the range of valid array indices.

The third category of dynamic error is a dynamic memory manipulation error using a pointer memory variable. This occurs when memory pointers no longer reflect the actual layout of memory due to problems with dynamic memory manipulation. This often involves a "dangling pointer" which is a memory pointer which points to a block of memory that has since been "freed," that is, deallocated. While the memory pointer still points to the address of the same freed memory block, the address is no longer a

representative of the dynamic state associated with the original pointer assignment. Six types of errors can occur, such as reading from or writing to a dangling pointer, passing a dangling pointer as an argument of a function, returning a dangling pointer from a function, freeing the same memory block multiple times, freeing stack memory (local variables), and attempting to free a memory block using a pointer that does not point to the beginning of a valid memory block.

Referring to FIG. **14**, a source code listing of a computer program containing a dynamic memory manipulation error using a pointer memory variable is shown. A pointer memory variable "ptr" is defined (line **3**). Next, a 10-character memory block is allocated and its pointer assigned to pointer memory variable "ptr" (line **4**). The pointer memory variable "ptr" is incremented (line **5**) and an attempt is made to free the memory block that it points to (line **6**). However, the attempt can ultimately lead to memory corruption since pointer memory variable "ptr" no longer points to the start of the memory block that was originally assigned to it. Therefore, a dynamic memory manipulation error occurs.

Referring to FIG. **15**, a diagram illustrating a parse tree representation of the source code listing in FIG. **14** is shown. The function declaration (lines **1**, **2** and **8**) is represented by function declaration node **140**. The declaration of pointer memory variable "ptr" (line **3**) is represented by variable declaration node **141**. The allocation of the 10-character memory block is represented by call expression node **147**, which has two operands, a function declaration and an argument, represented respectively by nodes **148** and **149**. The result from this function call is assigned to the pointer memory variable (line **4**), which is represented by assignment node **142**. This node has two operands, a pointer memory variable and the function call, represented respectively by nodes **146** and **147**. The pointer increment operation (line **5**) is represented by node **143**. The free memory block operation (line **6**) is represented by node **144**, which has two operands: a function call declaration and an argument, represented respectively by nodes **150** and **151**. The return operation (line **7**) is represented by node **145**.

Referring to FIGS. **16A** and **16B**, a flow chart for a routine for detecting a dynamic memory manipulation error using a pointer memory variable error condition is shown. This involves a more general error check than for the six specific types of memory manipulation errors listed above and is therefore presented initially.

First, a flow analysis is performed on the source code to identify a declaration operation for a pointer memory variable (block **160**), comprising an identifier operand for identifying the variable. If a declaration operation is found (block **161**), instrumentation nodes are inserted after the parse tree node corresponding to the declaration operation (block **162**). These nodes are for storing in a pointer record a value field for a memory address contained in the pointer memory variable during execution. Initially, the pointer memory variable points to nothing and the pointer record is therefore empty.

A flow analysis is then performed on the source code to identify a memory allocation operation for allocating a memory block to the pointer memory variable being checked (block **163**). If a memory allocation operation is found (block **164**), instrumentation nodes are inserted into the parse tree after the nodes corresponding to the memory allocation operation (block **165**). These nodes are for storing an allocation record for use by the error-checking engine during execution. Each allocation record contains the following information: block size, starting memory address for the block, addresses of memory pointers that point to the memory block, a list of memory pointers that are contained within the memory block, and state information regarding the memory block.

Next, a flow analysis is performed on the source code to identify an assignment operation to the selected pointer memory variable (block **166**). If an assignment operation is found (block **167**), instrumentation nodes are inserted into the parse tree after the nodes corresponding to the assignment operation (block **168**). These nodes are for indicating to the error-checking engine that the pointer memory variable may contain a different and possibly invalid memory address.

The previous steps having been accomplished, the routine can therefore perform a dynamic memory manipulation check (block **169**). Referring to FIGS. **17A** and **17B**, a flow chart for a routine for performing a dynamic memory manipulation check is shown. This performs the six types of dynamic memory manipulation checks listed previously.

A flow analysis is performed on the source code to identify a read operation or a write operation using the pointer memory variable being checked (block **180**). If a read operation or a write operation is found (block **181**), instrumentation nodes are inserted into the parse tree before the nodes corresponding to the read operation or the write operation (block **182**). During execution, the error-checking engine can indicate that the read operation or the write operation is attempting to operate on a pointer memory variable when it contains a dangling pointer, this is, a memory address for a freed memory block.

Next, a flow analysis is performed on the source code to identify a function call operation using the pointer memory variable being checked (block **183**). If a function call operation is found (block **184**), instrumentation nodes are inserted into the parse tree before the nodes corresponding to the function call operation (block **185**). During execution, the error-checking engine can indicate that the function call operation is calling a function using a pointer memory variable containing a memory address for a freed memory block.

Next, a flow analysis is performed on the source code to identify a function call return operation using the pointer memory variable being checked (block **186**). If a function call return operation is found (block **187**), instrumentation nodes are inserted into the parse tree before the nodes corresponding to the function call return operation (block **188**). During execution, the error-checking engine can indicate that the function call return operation is returning a memory address for a freed memory block to the calling function in the computer program.

Finally, a flow analysis is performed on the source code to identify a free memory block operation using the pointer memory variable being checked (block **189**). If a free memory block operation is found (block **190**), instrumentation nodes are inserted into the parse tree before the nodes corresponding to the free memory block operation (block **191**). During execution, the error-checking engine can indicate that the free memory block operation is attempting to free a memory block multiple times by using a pointer memory variable pointing to an already freed memory block or is attempting to free a stack frame (local variables) or is attempting to free a memory block when the memory address does not equal the starting memory address of the memory block.

Referring to FIG. 18, a diagram illustrating an instrumented parse tree representation of the source code listing in FIG. 14 is shown. The instrumentation nodes for the declaration operation are inserted as declare node 200. The instrumentation nodes for the assignment operation are inserted as pointer assignment node 201. Similarly, the instrumentation nodes for a further assignment operation are inserted as reassign node 202. Finally, the instrumentation nodes for the pre-memory block operation inserted as check arg to deallocate node 203. Declaration node 200 indicates to the error-checking engine that the pointer memory variable "ptr" is declared and uninitialized. The pointer assignment node 200 and reassign node 202 indicate to the error-checking engine that the pointer memory variable "ptr" has been initialized and incremented, respectively. The check arg to deallocate node 203 indicates to the error-checking engine that the program is attempting to free the memory pointed to by the pointer memory variable "ptr."

The fourth category of dynamic error is an inappropriate use of a pointer memory variable. Five types of errors can occur, comprising a pointer operation on a null pointer, a pointer operation on an uninitialized pointer, a pointer operation on a pointer that does not point to valid data, a pointer operation attempting to compare or otherwise relate memory pointers that fail to point to the same type of data object, and an attempt to make a function call using a function pointer that does not point to a function.

Referring to FIG. 19, a source code listing of a program segment containing an inappropriate use of a pointer memory variable is shown. Two long integer pointers "a" and "b" and a pointer to a function returning a long integer "foo" are defined (line 1). Next, the memory address of variable "a" is assigned using a cast to a pointer to a function returning a long integer "foo" (line 2). Finally, the return value of a function call to "foo" is assigned to variable "b" (line 3). This code segment is problematic because the function pointer "foo" actually points to a location in the program stack representing the memory block assigned to variable "a" instead of an appropriate entry point in the code segment. Therefore, the function pointer "foo" has been inappropriately used and a pointer memory variable error condition occurs.

Referring to FIG. 20, a diagram illustrating a parse tree representation of the source code listing in FIG. 19 is shown. The long integer variable declarations (line 1) are represented by variables declared node 210. The assignment operation (line 2) is represented by node 211, which has two operands, a variable (representing the left-hand side of the assignment) and an address expressions (representing the right-hand side of the assignment), represented respectively by nodes 212 and 213. The address expression node 213 operates on variable "a", which is represented by variable node 214. The assignment node 211 is followed by assignment node 215, which represents the assignment to variable "b" (line 3). This node has two operands, a variable and a function call to "foo", represented respectively by nodes 216 and 217. The call expression node 217 has one operand, a variable, represented by node 218.

Referring to FIGS. 21A and 21B, a flow chart for a routine for an inappropriate use of a pointer memory variable error condition is shown. First, a flow analysis is performed on the source code to identify a declaration operation for a pointer memory variable (block 230), comprising an identifier operand for identifying the variable. If a declaration operation is found (block 231), instrumentation nodes are inserted after the parse tree node corresponding to the declaration operation (block 232). These nodes are for storing in a pointer record a value field for a memory address contained in the pointer memory variable during execution. Initially, the pointer memory variable points to nothing and the pointer record is therefore empty.

Next, a flow analysis is performed on the source code to identify an operation using the pointer memory variable being checked (block 233). If an operation is found (block 234), instrumentation nodes are inserted into the parse tree before the nodes corresponding to the operation (block 235). These nodes are for performing the five types of error checks listed above, including checking for operations on a null pointer, an uninitialized pointer, a pointer to invalid data, mismatched pointer types, and an invalid function call pointer.

Referring to FIG. 22, a diagram illustrating an instrumented parse tree representation of the source code listing in FIG. 19 is shown. The instrumentation node for the function pointer declaration operation is inserted as declare function "ptr" node 220. The instrumentation node for the function pointer check is inserted as func_ptr_check node 221. The declare function "ptr" node 220 is used by the error-checking engine for runtime pointer tracking. The func_ptr_check node 221 is the actual check for a bad function pointer. During runtime, the error-checking engine determines that the value assigned to the function pointer "foo", is an address on the stack and is not an appropriate function address.

The fifth category of dynamic error is a memory leak error, which occurs when a dynamically allocated memory block is no longer referenced by a memory pointer and consequently can never be freed (deallocated). There are three types of errors. The first, a leak while freeing memory, can occur when a block of memory is freed which contains memory pointers that point to other allocated memory blocks. Any references to those allocated memory blocks are lost. The second, a leak return value, occurs when a function call returns an allocated memory block but the calling function does not assign that memory block to a pointer memory variable. The third, leak scope, occurs when a local pointer memory variable points to a memory block that is also local in scope and the function does not free the memory which it uniquely references before it goes out of scope.

Referring to FIG. 23, a source code listing of a computer function containing a memory leak error is shown. A 10-character memory block is allocated and its pointer assigned to pointer memory variable "ptr," which is a local variable defined in the same statement (line 3). The function "foo" uniquely references the memory block allocated whose memory address is assigned to the local pointer memory variable "ptr." The function returns (line 4) with "ptr" going out of scope. Consequently, the memory block formerly pointed to by "ptr" is leaked since "ptr" is no longer accessible. Therefore, a memory leak error occurs.

Referring to FIG. 24, a diagram illustrating a parse tree representation of the source code listing in FIG. 23 is shown. The function declaration (lines 1, 2 and 5) is represented by function declaration node 240. The declaration of local pointer memory variable "ptr" (line 3) is represented by a variable declaration node 241. Similarly, the allocation of the 10-character memory block is represented by call expression node 245, which has two operands, an argument and a function declaration, represented respectively by nodes 246 and 247.

The function declaration calls a memory allocation routine for dynamically allocating a block of memory. Such a

routine could be the "malloc( )" function call or the like in C language. The result from this routine is assigned to the local pointer memory variable (line **3**), which is represented by assignment node **242**. This node has two operands, a pointer memory variable and the function call, represented respectively by nodes **244** and **245**. The return operation (line **4**) is represented by node **243**.

Referring to FIG. **25**, a flow chart for a routine for detecting a memory leak error condition is shown. First, a flow analysis is performed on the source code to identify a declaration operation for a pointer memory variable (block **250**), including an identifier operand for identifying the variable. If a declaration operation is found (block **251**), instrumentation nodes are inserted after the parse tree node corresponding to the declaration operation (block **252**). These nodes store a pointer record indicating information about the block of memory that it points to.

Next, a flow analysis is performed on the source code to identify an exit from scope operation, such as a return from a function call (block **253**). If such an operation is found (block **254**), instrumentation nodes are inserted before the parse tree node corresponding to the exit from scope operation (block **255**). These nodes are for detecting memory leaks. Thus, when the function exits or the pointer goes out of scope, the error-checking engine is informed by a "pop scope" directive. Upon that occurrence, the engine can examine the list of pointers declared in that scope. For each pointer, if the block of memory that it is pointing to is only pointed to by a local pointer variable, the memory is leaked when the pointer goes out of scope.

Memory leaks can be detected in one of two ways. The first is during an assignment of a new address to a pointer variable. If the memory block that used to be pointed to by the pointer is being reassigned and the memory block is only pointed to by that pointer, the block is leaked by the assignment operation. Second, a memory leak can occur upon the exiting of a scope. If there is a memory block which is pointed to only by a pointer declared locally in scope within the function being exited, the memory block is leaked.

During operation, the error-checking engine initializes a pointer record for each pointer in a function upon activation. For any assignment of an address to a pointer, the pointer record is updated to indicate that the pointer contains the address of an allocated memory block. Similarly, the memory block record pointer list is updated to indicate that the pointer is pointing to that block. Finally, upon the exit from the routine, all pointer records are cleared.

Referring to FIG. **26**, a diagram illustrating an instrumented parse tree representation of the source code listing in FIG. **23** is shown. The assignment of the local allocated memory block to the local pointer memory variable in assignment node **242** indicates to the error-checking engine that variable "ptr" is pointing to a particular memory block. Subsequently, when the pointer goes out of scope, the pointer record maintained by the error-checking engine is removed from the memory block record. Since that record now has an empty list of pointers pointing to it, the error-checking engine can detect that memory has been leaked. The instrumentation nodes for the declaration operation are inserted as declare local pointer node **270**. The instrumentation nodes for the pop scope operation are inserted as pop scope node **271**. Declare local pointer node **270** indicates to the error-checking engine that the local pointer memory variable "ptr" is declared and uninitialized. The pop scope node **271** indicates to the error-checking engine that "ptr" has gone out of scope due to a return operation from the function.

Instrumentation routines can also be used to introduce a user-definable instrumentation routine known as an interface. This type of routine enables a user to add their own rules for transforming the source code. An interface routine can have the same behavior as the source code it is replacing or it can do something completely different, such as checking values of variables, simulating errors or performing any other type of dynamic tasks.

Typically, user-defined interfaces allow the user to add custom error checking to function calls as a means of enforcing rules on the way that the function is called and the side effects that it has on memory. These types of interfaces check that all parameters are of the correct data type, that memory pointers point to memory blocks of the appropriate size, and that each parameter value is within its correct range.

Referring to FIG. **27A**, a program segment of a computer function to be instrumented with an interface is shown. Here, the interface is for a memory allocation call using the "malloc( )" function (line **1**). Referring to FIG. **27B**, the interface routine is shown. In this example, the interface is similar to a complicated macro definition because the given function call is replaced by a user defined interface.

The "iic_" prefixed functions are expanded into function calls to the runtime back-end processor. Two such function calls are employed in this example. The "iic_error ( )" function call communicates error messages to the back-end processor (lines **5** and **10**). The "iic_alloc( )" function call communicates to the back-end processor that a block of memory of size "size" has been allocated and is pointed to by pointer "a" (line **8**).

In addition, two further error checks are performed by the interface. First, it checks to see whether the size of the memory block being allocated is a positive number (lines **4–5**). If it is, the memory allocation "malloc( )" function call is allowed to go forward (line **6**). Next, the pointer memory variable "a" is checked to determine if the memory allocation function call failed, and if so, the back-end processor is so informed (lines **7–10**).

Referring to FIG. **28**, a diagram illustrating a parse tree representation of the program segment shown in FIG. **27A** is shown. The assignment operation (line **1**) is represented by assignment node **280**, which has two operands, a variable operand (representing the left-hand side of the assignment) and a function call expression operand (representing the right-hand side of the assignment), represented respectively by nodes **281** and **282**. In turn, the call expression node **282** has two operands, a function declaration for a "malloc( )" function and an integer constant, represented respectively by nodes **283** and **284**.

Referring to FIG. **29**, a flow chart for a routine for inserting an interface is shown. Before an interface can be used, it must first be pre-processed to convert it from source code into an intermediate form and then stored in a database for later use (block **29**). A flow analysis is then performed on the source code to identify function calls having a corresponding interface description to that stored in the database (block **291**). If a matching function call is found (block **292**), the interface is inserted into the parse tree in a multi-step process.

This process includes the step of first removing the existing function call from the parse tree (block **293**). Next, the stored intermediate form for the corresponding interface is read from the database (block **294**). The stored intermediate form is inserted as interface nodes in the parse tree in place of the node corresponding to the original function call

(block **295**). The original function call arguments are substituted into placeholders in the interface nodes (block **296**). This enables the interface to actually perform the original function which is called within the interface function itself. Finally, the return statement and the interface node are replaced with an assignment of the result of the interface routine to the actual call to the original function call (block **297**). This enables the original calling function to receive the result that was expected without the interface.

Referring to FIG. **30**, a diagram illustrating an instrumented parse tree representation of the program segment shown in FIG. 27A is shown. This parse tree differs from those used in other parts of the invention. The original parse tree node representation has been broken into two sections and grafted onto the intermediate form for the interface routine. The entire parse tree shown in FIG. **30** is grafted in place of the parse tree shown in FIG. **28**.

The grafted parse tree segment shown in FIG. **30** reflects the program structure of the interface routine source code. The conditional check for a positive memory block size is inserted as "if conditional" node **300** with the conditional test represented by node **301** (line **4**). The body of the conditional statement is represented by error notify node **302** and error string node **303** (line **5**) which communicate to the runtime back-end processor the occurrence of a dynamic runtime error condition. The original function call to the memory allocation routine "malloc( )" (line **6**) is represented by assignment node **304** which has two operands, a variable and an expression call, represented respectively by nodes **305** and **282**. Note that nodes **282**, **283** and **284** are substituted into the interface intermediate form in the place of placeholders.

The error check for a memory allocation operation failure is represented by "if conditional" node **306** with the conditional test represented by node **307** (line **7** and **9**). The body of the "then" condition is represented by allocation notify node **308** (line **8**) which tells the runtime processor that a block of memory of size "size" has been allocated and is pointed to by pointer "a". The body of the "else" condition is represented by error notify node **309** (line **10**) which has one operand, error string node **310**. Finally, an assignment condition is grafted to the end of the intermediate form to assign the result from the interface, represented by variable node **311**, to the original function call.

Instrumentation routines can also be used to insert support for automatic test case generation. One embodiment of the present invention is described in "Overview of the Design of TGS System," which is attached as Appendix B, the subject matter of which is hereby incorporated by reference as if set forth in full. By performing a flow analysis of the source code, a two-fold criteria can be satisfied. First, instrumentation routines can be inserted to automatically generate program inputs to achieve full testing of all flow paths in the executable program. Second, instrumentation routines can be used to identify inputs that cause the program to perform incorrectly.

The method involves analyzing the source code to identify points where input data is needed. Next, various techniques are employed, ranging from random number generation to heuristic flow analysis techniques, to generate a set of input cases that satisfy the two-fold criteria stated above. The resulting executable program is linked to a test harness which repeatedly runs the program with different input values and adds unique test cases to a database of test case data. The testing algorithm converges when the two-fold criteria is met or when no new test cases can be generated in a reasonable amount of time.

Referring to FIG. **31**, a source code listing of a computer function to be instrumented for automatic test case generation is shown. The purpose of this function is to accept an input character and determine whether it is an integer. Three character memory variables are declared, "b," "c" and pointer "ptr" (line **3**). An input function "getchar( )" is called to obtain an input character whose value is assigned to variable "c" (line **4**). That value is checked to determine whether it falls in the numeric range of ASCII codes for integer characters (line **5**). If it does not, the pointer memory variable "ptr" is set to 0 (line **6**). The variable "b" is assigned the difference of a "0" ASCII character code subtracted from the input character stored in variable "c" (line **8**). A problem with this function is that the pointer memory variable "ptr" is not yet initialized before it is dereferenced by setting it to 0.

Referring to FIG. **32**, a diagram illustrating a parse tree representation of the source code listing in FIG. **31** is shown. The function declaration (lines **1**, **2** and **9**) is represented by function declaration node **500**. The declaration of variables "a" "b" and "ptr" (line **3**) is represented by variables declared node **501**. The assignment of the input character is represented by assignment node **502**, which has two operands, a variable and a function call, represented respectively by nodes **503** and **504**. The function call, represented by call expression node **504**, has a single operand, function decl node **505** which contains the identifier for the "getchar( )" function. The conditional statement (line **5**) is represented by if stmt node **506**, which points to a node containing the conditions to be tested, represented by node **507**. The body of the condition node contains an assignment statement (line **6**) represented by assignment node **508**, which has two operands, a pointer reference and an integer, respectively represented by nodes **509** and **510**.

Referring to FIG. 33, a diagram illustrating an instrumented parse tree representation of the source code listing in FIG. **31** is shown. The instrumentation nodes for the declaration operation are inserted as declare node **520**. The instrumentation nodes for the character input generation operation are inserted as input generator node **521**. The instrumentation nodes for the uninitialized pointer check operation are inserted as check pointer node **522**. Declare node **520** indicates to the error-checking engine that the local pointer memory variable "ptr" is declared and uninitialized. The input generator node **521** indicates to the error-checking engine that the function call to "getchar( )" is replaced by a test case generator function which generates random inputs between 0 and 255. The check pointer node **522** indicates to the error-checking engine that the function is attempting to use a pointer memory variable that may not have been initialized previously and is probably pointing to an invalid memory address.

As will be realized, the present invention is capable of other and different embodiments and its several details are capable of modifications in various respects, all without departing from the spirit and scope of the present invention. Accordingly, the drawings and detailed description of the preferred embodiment are to be regarded as illustrative in nature and not as restrictive.

What is claimed is:

1. A method using a computer for instrumenting in real time a computer program source code to facilitate the detection of runtime errors, the computer program being represented by a parse tree, each such runtime error having at least one instrumentation rountine for communicating an occurence of the associated runtime error to an error-checking engine, each such instrumentation routine being represented by a parse tree fragment, comprising the steps of:

performing a real time flow analysis on the parse tree for the computer program to determine an appropriate instrumentation routine for detecting the runtime errors;

instrumenting in real time the parse tree for the computer program by grafting the parse tree fragment for each such instrumentation routine onto the parse tree for the computer program;

generating executable object code from the grafted parse tree containing references to the instrumentation routines; and

executing the generated object code on a computer by transferring the runtime errors to the error-checking engine and linking the error-checking engine with the computer program to determine errors.

2. A method according to claim 1, wherein the runtime error comprises a read operation to an uninitialized memory variable, further comprising the steps of:

retrieving a memory address for a memory variable from a stack frame represented as interconnected nodes stored in the parse tree and auguementing the parse tree with the instrumentation nodes for indicating that the memory variable is possibly uninitialized;

identifying such a read operation to the memory address and augmenting the parse tree with the instrumentation nodes for indicating that the possibly unintialized memory variable is being read by the read operation; and

identifying a write operation to the memory address and augmenting the parse tree with the instrumentation nodes for indicating that the possibly unintialized memory variable is initialized.

3. A method according to claim 1, wherein the runtime error comprises a write operation to an invalid memory address for a complex memory variable, further comprising the steps of:

identifying a declaration operation of the complex memory variable comprising an identifier operand and a dimension operand that defines a memory address range and augmenting the parse tree with the instrumentation nodes for storing the identifier operand and the dimension operand; and

identifying such a write operation using the complex memory variable and augmenting the parse tree with instrumentation nodes for indicating an access by the write operation to at least one of such a memory address indentified by the indentifier operand that is invalid or such a memory address that falls outside of the memory address range defined by the dimension operand.

4. A method according to claim 1, wherein the runtime error comprises a runtime memory allocation operation using a pointer memory variable, further comprising the steps of:

identifying a declaration operation for such a pointer memory variable comprising an indentifier operand and augmenting the parse tree with instrumentation nodes for storing a pointer record for the pointer memory variable and a corresponding memory address;

identifying such a runtime memory allocation operation for allocating a memory block to the pointer memory variable and augmenting the parse tree with instrumentation nodes for storing an allocation record for the memory block indentified by the indentifier operand;

identifying an assignment operation assigning the corresponding memory address to the pointer memory variable and augmenting the parse tree with instrumentation nodes for storing the corresponding memory address into the pointer record; and

augmenting the parse tree with instrumentation nodes for performing a runtime memory manipulation check using the pointer record and the allocation record.

5. A method according to claim 1, wherein the runtime error comprises an inappropriate use of a pointer memory variable, further comprising the steps of:

identifying a declaration operation for such a pointer memory variable and augmenting the parse tree with instrumentation nodes for storing a pointer record for the pointer memory variable and a corresponding memory address;

identifying an operation using the pointer memory variable and augmenting the parse tree with instrumentation nodes for indicating whether at least one of the operation is using the pointer memory variable when the corresponding memory address is equal to null, the operation is using the pointer memory variable when the corresponding memory address is unintialized, the operation is using the corresponding memory address not pointing to valid data, the operation is comparing the pointer memory variable not pointing to identical types of data or the operation is making a function call using the pointer memory variable not containing a valid function address.

6. A method according to claim 1, wherein the runtime error comprises a memory leak, further comprising the steps of:

identifying a declaration operation for a pointer memory variable and augmenting the parse tree with instrumentation nodes for storing a pointer record for the pointer memory variable; and

identifying an exit from scope operation and augmenting the parse tree with instrumentation nodes for checking for memory leaks using the scope operation and the pointer record.

7. A system for instrumenting a computer program to facilitate the detection of runtime errors, the computer program being represented by a parse tree, each such runtime error having at least one instrumentation routine for communicating an occurrence of the associated runtime error to an error-checking engine, each such instrumentation routine being represented by a parse tree fragment, comprising:

means for performing a real time flow analysis on the parse tree for the computer program to determine an appropriate instrumentation routine for detecting the runtime errors;

means for instrumenting in real time the parse tree for the computer program by grafting the parse tree fragment for each such instrumentation routine onto the parse tree for the computer program;

means for generating executable object code from the grafted parse tree containing references to the instrumentation routines; and

means for executing the generated object code by transferring the runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

8. A method using a computer for instrumenting an intermediate representation of a computer program for dynamic debugging, comprising the steps of:

creating in real time the intermediate representation of the computer program in the computer by translating

source code comprising the computer program into an equivalent intermediate form;

analyzing in real time the intermediate representation of the computer program for instruction flow for determining an appropriate type of instrumentation for use in dynamic debugging;

augmenting in real time the intermediate representation of the computer program with at least one further intermediate representation wherein the at least one further intermediate representation comprises the appropriate type of instrumentation translated into the equivalent intermediate form;

generating executable object code from the augmented intermediate representation of the computer program containing references to the instrumentation routines and storing the executable code in the computer; and

executing the executable object code by transferring runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

9. A method according to claim 8, wherein the step of creating the intermediate representation of the computer program further comprises the step of:

building a parse tree as the equivalent intermediate form.

10. A method according to claim 9, further comprising the steps of:

lexically analyzing the source code of the computer program to form a sequence of tokens;

parsing the sequence of tokens to form grammatical phrases;

semantically analyzing the grammatical phrases to form the parse tree.

11. A method according to claim 8, wherein the step of augmenting the intermediate representation of the computer program further comprises the steps of:

inserting such intermediate representations of instrumentation for checking functionality of the source code of the computer program and communicating the functionality to the error-checking engine; and

inserting such intermediate representations of instrumentation for communicating non-functional information regarding the source code of the computer program to the error-checking engine.

12. A method according to claim 11, further comprising the steps of:

communicating at least one of a declaration of variables, pointer addresses and pointer sizes;

communicating assignments of pointers;

communicating at least one of function entry and exit point indicators; and

communicating memory allocation indicators.

13. A method according to claim 11, further comprising the steps of:

detecting a read operation to an uninitialized memory variable;

detecting a write operation to an invalid memory address for a complex memory variable;

detecting a dynamic memory manipulation error using a pointer memory variable;

detecting an inappropriate use of a pointer memory variable;

detecting a memory leak error;

inserting a user-defined instrumentation routine; and

inserting an automatic test case generation routine.

14. A method using a computer for instrumenting an intermediate representation of a computer program for dynamic debugging, comprising the steps of:

creating the intermediate representation of the computer program in the computer by translating source code comprising the computer program into an equivalent intermediate form consisting of at least an additional sequence of code instructions;

analyzing in real time the additional sequence of code instructions to determine an appropriate type of instrumentation for use in the dynamic debugging;

augmenting in real time the additional sequence of code instructions with further code instructions to invoke the appropriate type of instrumentation;

generating executable object code containing references to the instrumentation routines using the augmented sequence of additional code and storing the executable code in the computer, and

executing the executable code by transferring runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

15. A method according to claim 14 wherein the appropriate type of instrumentation includes code directed to detecting a read operation to an uninitialized memory variable.

16. A method according to claim 14 wherein the appropriate type of instrumentation includes code directed to detecting a write operation to an invalid memory address for a complex memory variable.

17. A method according to claim 14 wherein the appropriate type of instrumentation includes code directed to detecting an inappropriate use of a pointer memory variable.

18. A method according to claim 14 wherein the appropriate type of instrumentation includes code directed to detecting dynamic memory manipulation error using a pointer memory variable.

19. A method according to claim 14 wherein the appropriate type of instrumentation includes code directed to detecting a memory leak error.

20. A method according to claim 14 wherein the appropriate type of instrumentation includes code corresponding to a user defined instrumentation routine.

21. A method according to claim 14 wherein the appropriate type of instrumentation includes code corresponding to an automatic test case generation routine.

22. A method using a computer for dynamic debugging, comprising the steps of:

initiating compilation of a computer program;

interrupting the compilation of a computer program in order to capture the intermediate representation of the computer program created by the compilation process;

augmenting in real time the intermediate representation of the computer program to add instrumentation for use in dynamic debugging;

reinitiating the compilation process to generate executable object code containing references to the instrumentation routines;

executing the object code by transferring runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

23. A method using a computer for instrumenting in real time a computer program source code to facilitate the

detection of runtime errors, the computer program being represented by a parse tree, each such runtime error having at least one instrumentation routine for communicating an occurrence of the associated runtime error to an error-checking engine, each such instrumentation routine being represented by a parse tree fragment, comprising the steps of:

performing a real time flow analysis on the parse tree for the computer program to determine an appropriate instrumentation routine for detecting the runtime errors;

instrumenting in real time the parse tree for the computer program by grafting the parse tree fragment for each such instrumentation routine onto the parse tree for the computer program;

generating executable object code from the grafted parse tree containing references to the instrumentation routines; and

continuously executing the generated object code on the computer by transferring runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

24. A method according to claim 23, wherein the runtime error comprises a read operation to an uninitialized memory variable, further comprising the steps of:

retrieving a memory address for a memory variable from a stack frame represented as interconnected nodes stored in the parse tree and augmenting the parse tree with the instrumentation nodes for indicating that the memory variable is possibly uninitialized;

identifying such a read operation to the memory address and augmenting the parse tree with the instrumentation nodes for indicating that the possibly uninitialized memory variable is being read by the read operation; and

identifying a write operation to the memory address and augmenting the parse tree with the instrumentation nodes for indicating that the possibly uninitialized memory variable is initialized.

25. A method according to claim 23, wherein the runtime error comprises a write operation to an invalid memory address for a complex memory variable, further comprising the steps of:

identifying a declaration operation for the complex memory variable comprising an identifier operand and a dimension operand that defines a memory address range and augmenting the parse tree with the instrumentation nodes for storing the identifier operand and the dimension operand; and

identifying such a write operation using the complex memory variable and augmenting the parse tree with instrumentation nodes for indicating an access by the write operation to at least one of such a memory address identified by the identifier operand that is invalid or such a memory address that falls outside of the memory address range defined by the dimension operand.

26. A method according to claim 23, wherein the runtime error comprises a runtime memory allocation operation using a pointer memory variable, further comprising the steps of:

identifying a declaration operation for such a pointer memory variable comprising a identifier operand and augmenting the parse tree with instrumentation nodes for storing a pointer record for the pointer memory variable and a corresponding memory address;

identifying such a runtime memory allocation operation for allocating a memory block to the pointer memory variable and augmenting the parse tree with instrumentation nodes for storing an allocation record for the memory block identified by the identifier operand;

identifying an assignment operation assigning the corresponding memory address to the pointer memory variable and augmenting the parse tree with instrumentation nodes for storing the corresponding memory address into the pointer record; and

augmenting the parse tree with instrumentation nodes for performing a runtime memory manipulation check using the pointer record and the allocation record.

27. A method according to claim 23, wherein the runtime error comprises an inappropriate use of a pointer memory variable, further comprising the steps of:

identifying a declaration operation for such a pointer memory variable and augmenting the parse tree with instrumentation nodes for storing a pointer record for the pointer memory variable and a corresponding memory address;

identifying an operation using the pointer memory variable and augmenting the parse tree with instrumentation nodes for indicating whether at least one of the operation is using the pointer memory variable when the corresponding memory address is equal to null, the operation is using the pointer memory variable when the corresponding memory address is uninitialized, the operation is using the corresponding memory address not pointing to valid data, the operation is comparing the pointer memory variable not pointing to identical types of data or the operation is making a function call using the pointer memory variable not containing a valid function address.

28. A method according to claim 23, wherein the runtime error comprises a memory leak, further comprising the steps of:

identifying a declaration operation for a pointer memory variable and augmenting the parse tree with instrumentation nodes for storing a pointer record for the pointer memory variable; and

identifying an exit from scope operation and augmenting the parse tree with instrumentation nodes for checking for memory leaks using the scope operation and the pointer record.

29. A system for instrumenting in real time a computer program source code to facilitate the detection of runtime errors, the computer program being represented by a parse tree, each such runtime error having at least one instrumentation routine for communicating an occurrence of the associated runtime error to an error-checking engine, each such instrumentation routine being represented by a parse tree fragments comprising:

means for performing a real time flow analysis on the parse tree for the computer program to determine an appropriate instrumentation routine for detecting the runtime errors;

means for instrumenting in real time the parse tree for the computer program by grafting the parse tree fragment for each such instrumentation routine onto the parse tree for the computer program;

means for generating executable object code from the grafted parse tree containing references to the instrumentation routines; and

means for continuously executing the generated object code by transferring runtime errors to an error-checking

engine and linking the error-checking engine with the computer program to determine errors.

30. A method using a computer for instrumenting in real time an intermediate representation of a computer program for dynamic debugging, comprising the steps of:

creating the intermediate representation of the computer program in the computer by translating source code comprising the computer program into an equivalent intermediate form;

analyzing in real time the intermediate representation of the computer program for instruction flow for determining an appropriate type of instrumentation for use in dynamic debugging;

augmenting in real time the intermediate representation of the computer program with at least one further intermediate representation wherein the at least one further intermediate representation comprises the appropriate type of instrumentation translated into the equivalent intermediate form;

generating executable object code containing references to the instrumentation routines from the augmented intermediate representation of the computer program and storing the executable code in the computer; and

continuously executing the generated object code by transferring runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

31. A method according to claim 30, wherein the step of creating the intermediate representation of the computer program further comprises the step of:

building a parse tree as the equivalent intermediate form.

32. A method according to claim 31, further comprising the steps of:

lexically analyzing the source code of the computer program to form a sequence of tokens;

parsing the sequence of tokens to form grammatical phrases;

semantically analyzing the grammatical phrases to form the parse tree.

33. A method according to claim 30, wherein the step of augmenting the intermediate representation of the computer program further comprises the steps of:

inserting such intermediate representations of instrumentation for checking functionality of the source code of the computer program and communicating the functionality to the error-checking engine; and

inserting such intermediate representations of instrumentation for communicating non-functional information regarding the source code of the computer program to the error-checking engine.

34. A method according to claim 33, further comprising the steps of:

communicating at least one of a declaration of variables, pointer addresses and pointer sizes;

communicating assignments of pointers;

communicating at least one of function entry and exit point indicators; and

communicating memory allocation indicators.

35. A method according to claim 33, further comprising the steps of:

detecting a read operation to an uninitialized memory variable;

detecting a write operation to an invalid memory address for a complex memory variable;

detecting a dynamic memory manipulation error using a pointer memory variable;

detecting an inappropriate use of a pointer memory variable;

detecting a memory leak error;

inserting a user-defined instrumentation routine; and

inserting an automatic test case generation routine.

36. A method using a computer for instrumenting in real time an intermediate representation of a computer program source code for dynamic debugging, comprising the steps of:

creating the intermediate representation of the computer program in the computer by translating source code comprising the computer program into an equivalent intermediate form consisting of at least an additional sequence of code instructions;

analyzing in real time the additional sequence of code instructions to determine an appropriate type of instrumentation for use in the dynamic debugging;

augmenting in real time the additional sequence of code instructions with further code instructions to invoke the appropriate type of instrumentation;

generating executable object code containing references to the instrumentation routines using the augmented sequence of additional code and storing the executable code in the computer; and

continuously executing the executable code by transferring runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

37. A method according to claim 36 wherein the appropriate type of instrumentation includes code directed to detecting a read operation to an uninitialized memory variable.

38. A method according to claim 36 wherein the appropriate type of instrumentation includes code directed to detecting a write operation to an invalid memory address for a complex memory variable.

39. A method according to claim 36 wherein the appropriate type of instrumentation includes code directed to detecting an inappropriate use of a pointer memory variable.

40. A method according to claim 36 wherein the appropriate type of instrumentation includes code directed to detecting dynamic memory manipulation error using a pointer memory variable.

41. A method according to claim 36 wherein the appropriate type of instrumentation includes code directed to detecting a memory leak error.

42. A method according to claim 36 wherein the appropriate type of instrumentation includes code corresponding to a user defined instrumentation routine.

43. A method according to claim 36 wherein the appropriate type of instrumentation includes code corresponding to an automatic test case generation routine.

44. A method using a computer for dynamic debugging, comprising the steps of:

initiating compilation of a computer program;

interrupting the compilation of a computer program in order to capture the intermediate representation of the computer program created by the compilation process;

augmenting in real time the intermediate representation of the computer program to add instrumentation for use in dynamic debugging;

reinitiating the compilation process to generate executable object code containing references to the instrumentation routines;

continuously executing the object code by transferring runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

45. A method using a computer for instrumenting an intermediate representation of a computer program for dynamic debugging, comprising the steps of:

creating the intermediate representation of the computer program in the computer by translating source code comprising the computer program into an equivalent intermediate form;

analyzing in real time the intermediate representation of the computer program for instruction flow for determining an appropriate type of instrumentation for use in dynamic debugging;

augmenting in real time the intermediate representation of the computer program with at least one further intermediate representation wherein the at least one further intermediate representation comprises the appropriate type of instrumentation translated into the equivalent intermediate form;

generating executable object code containing references to the instrumentation routines from the augmented intermediate representation of the computer program and storing the executable code in the computer; and

executing the code executable in real time by transferring runtime errors to an error-checking engine and linking the error-checking engine with the computer program to determine errors.

46. A method according to claim 45, wherein the step of creating the intermediate representation of the computer program further comprises the step of:

building a parse tree as the equivalent intermediate form.

47. A method according to claim 46, further comprising the steps of:

lexically analyzing the source code of the computer program to form a sequence of tokens;

parsing the sequence of tokens to form grammatical phrases;

semantically analyzing the grammatical phrases to form the parse tree.

48. A method according to claim 45, wherein the step of augmenting the intermediate representation of the computer program further comprises the steps of:

inserting such intermediate representations of instrumentation for checking functionality of the source code of the computer program and communicating the functionality to the error-checking engine; and

inserting such intermediate representations of instrumentation for communicating non-functional information regarding the source code of the computer program to the error-checking engine.

49. A method according to claim 48, further comprising the steps of:

communicating at least one of a declaration of variables, pointer addresses and pointer sizes;

communicating assignments of pointers;

communicating at least one of function entry and exit point indicators; and

communicating memory allocation indicators.

50. A method according to claim 48, further comprising the steps of:

detecting a read operation to an uninitialized memory variable;

detecting a write operation to an invalid memory address for a complex memory variable;

detecting a dynamic memory manipulation error using a pointer memory variable;

detecting an inappropriate use of a pointer memory variable;

detecting a memory leak error;

inserting a user-defined instrumentation routine; and

inserting an automatic test case generation routine.

* * * * *

US005396631A

United States Patent [19]

Hayashi et al.

[11] Patent Number: 5,396,631

[45] Date of Patent: Mar. 7, 1995

[54] COMPILING APPARATUS AND A COMPILING METHOD

[75] Inventors: Masakazu Hayashi; Yutaka Igarashi; Masaaki Takiuchi; Kohichiro Hotta, all of Kawasaki, Japan

[73] Assignee: Fujitsu Limited, Kawasaki, Japan

[21] Appl. No.: 113,810

[22] Filed: Aug. 31, 1993

[30] Foreign Application Priority Data

Mar. 1, 1993 [JP] Japan 5-039841

[51] Int. Cl.[6] G06F 9/45

[52] U.S. Cl. 395/700; 364/DIG. 1; 364/280.5

[58] Field of Search .... 364/DIG. 1 MS File; 395/700

[56] References Cited

U.S. PATENT DOCUMENTS

4,667,290	5/1987	Goss et al.	395/700
4,763,255	8/1988	Hopkins et al.	395/700
5,146,594	9/1992	Iitsuka	395/700
5,193,190	3/1993	Janczyn et al.	395/700
5,212,794	5/1993	Pettis et al.	395/700
5,293,631	3/1994	Rau et al.	395/700

FOREIGN PATENT DOCUMENTS

2-81137 3/1990 Japan .

OTHER PUBLICATIONS

Computing Practices, *A Practical Tool Kit for Making Portable Compilers*, Andrew S. Tanenbaum et al., pp. 1-7, Apr. 1993.

Primary Examiner—Thomas M. Heckler
Attorney, Agent, or Firm—Staas & Halsey

[57] ABSTRACT

A compiling apparatus has a front end for providing intermediate representations according to a source program; an optimizing unit for optimizing the intermediate representations; an intermediate representation changing unit for changing the optimized intermediate representations; and a code output unit for providing code according to the lastly obtained intermediate representations. The compiling apparatus further has an optimizing structure determination unit for determining the number of repetitions of an optimization phase achieved by the optimizing unit and intermediate representation changing unit and selecting optimization functions carried out in each of the optimization phases. The intermediate representations are changed and optimized in each optimization phase according to the determination by the optimizing structure determination unit.

22 Claims, 19 Drawing Sheets

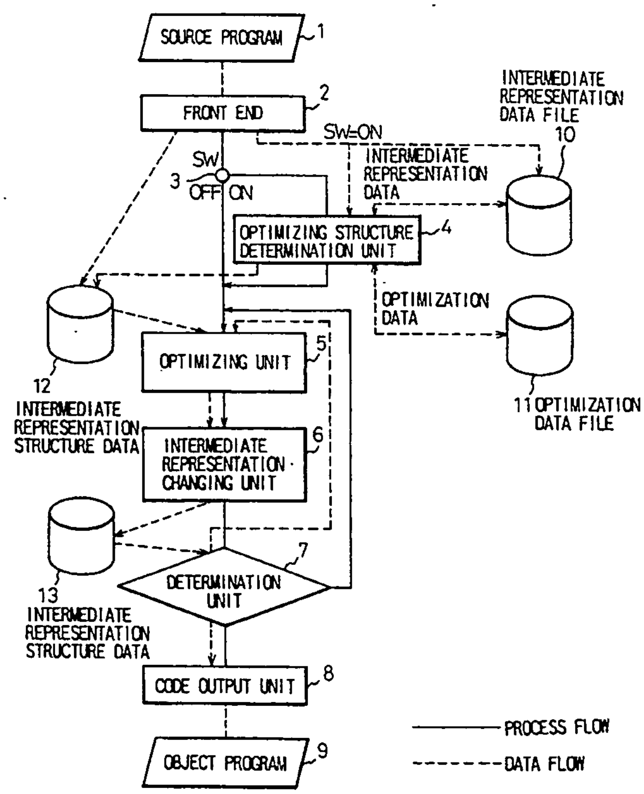


Fig. 1
PRIOR ART

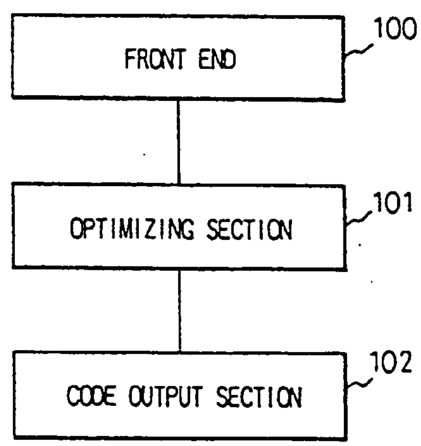


Fig. 2

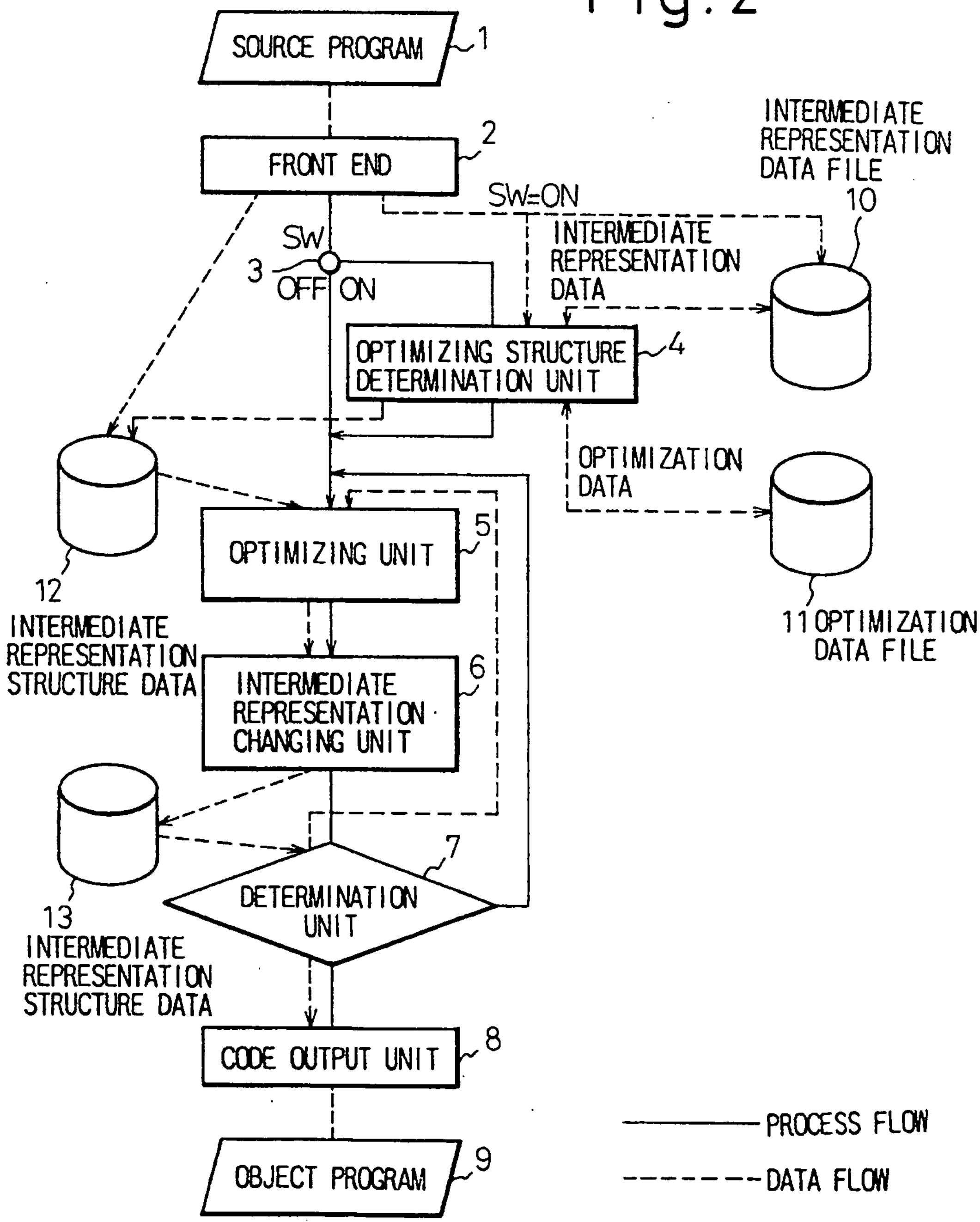

SOURCE PROGRAM
1
FRONT END
2
SW=ON
INTERMEDIATE REPRESENTATION DATA FILE
10
SW
3
OFF
ON
INTERMEDIATE REPRESENTATION DATA
OPTIMIZING STRUCTURE DETERMINATION UNIT
4
OPTIMIZATION DATA
OPTIMIZING UNIT
5
12
INTERMEDIATE REPRESENTATION STRUCTURE DATA
11 OPTIMIZATION DATA FILE
INTERMEDIATE REPRESENTATION CHANGING UNIT
6
DETERMINATION UNIT
7
13
INTERMEDIATE REPRESENTATION STRUCTURE DATA
CODE OUTPUT UNIT
8
OBJECT PROGRAM
9
PROCESS FLOW
DATA FLOW

Fig. 3

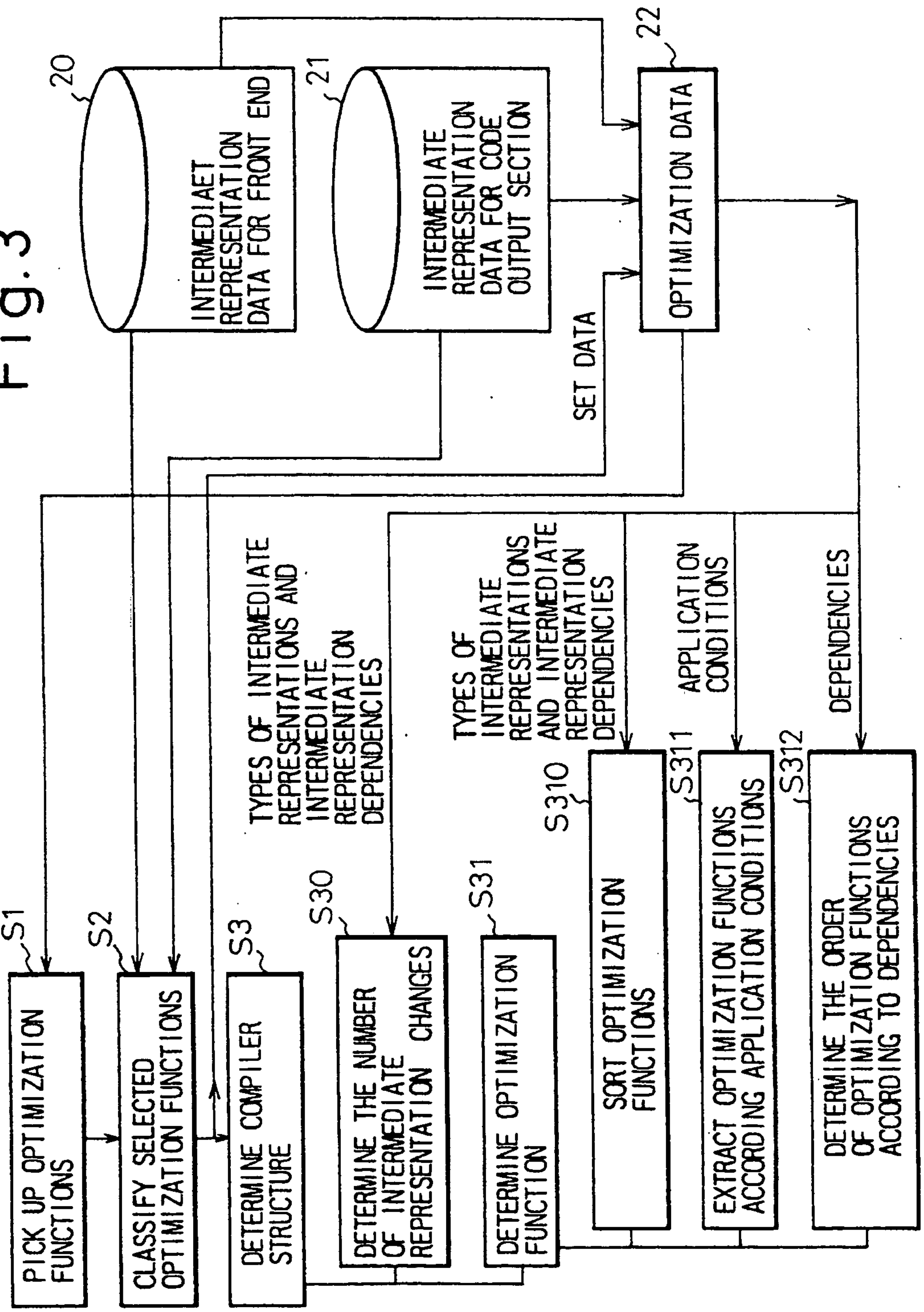

20
INTERMEDIAET REPRESENTATION DATA FOR FRONT END
21
INTERMEDIATE REPRESENTATION DATA FOR CODE OUTPUT SECTION
22
OPTIMIZATION DATA
SET DATA
S1
PICK UP OPTIMIZATION FUNCTIONS
S2
CLASSIFY SELECTED OPTIMIZATION FUNCTIONS
S3
DETERMINE COMPILER STRUCTURE
TYPES OF INTERMEDIATE REPRESENTATIONS AND INTERMEDIATE REPRESENTATION DEPENDENCIES
S30
DETERMINE THE NUMBER OF INTERMEDIATE REPRESENTATION CHANGES
S31
DETERMINE OPTIMIZATION FUNCTION
TYPES OF INTERMEDIATE REPRESENTATIONS AND INTERMEDIATE REPRESENTATION DEPENDENCIES
S310
SORT OPTIMIZATION FUNCTIONS
APPLICATION CONDITIONS
S311
EXTRACT OPTIMIZATION FUNCTIONS ACCORDING APPLICATION CONDITIONS
DEPENDENCIES
S312
DETERMINE THE ORDER OF OPTIMIZATION FUNCTIONS ACCORDING TO DEPENDENCIES

Fig.4(A)

```
--
TYPE:  LINDA (Generic)
CODE (OPERATOR):  LOAD
DATA TYPE:  i1, i2, i4, u1, u2, u4, r4, r8, r16, c8, c16, c32
OPERANDS
      1:  std, prg
      2:  axd, bxd, var
CONDITIONS
      BOUNDARY:  X (Unguaranteed)
      ADDRESS REPRESENTATION:  3
--
TYPE:  LINDA (Generic)
CODE (OPERATOR):  ADD
DATA TYPE:  i1, i2, i4, u1, u2, u4, r4, r8, r16, c8, c16, c32
OPERANDS
      1:  std, prg, var
      2:  axd, bxd, var
      3:  std, prg, var, cnt
CONDITIONS
      CONSTANT RANGE:  X (Unlimited)
--
TYPE:  LINDA (Generic)
CODE (OPERATOR):  cmove
DATA TYPE:  char
OPERANDS
      1:  axd, bxd
      2:  axd, bxd
CONDITIONS
--
```

Fig.4(B)

```
--
TYPE:  LINDA (Generic)
CODE (OPERATOR):  LOAD
DATA TYPE:  i1, i2, i4, u1, u2, u4, r4, r8
OPERANDS
      1:  std, prg
      2:  axd, bxd
CONDITIONS
      BOUNDARY:  axd, bxd (Boundaries for axd and bxd must be
guaranteed.)
      ADDRESS REPRESENTATION:  2 (Memory address with two values)
--
TYPE:  LINDA (Generic)
CODE (OPERATOR):  ADD
DATA TYPE:  i1, i2, i4, u1, u2, u4, r4, r8
OPERANDS
      1:  std, prg
      2:  std, prg
      3:  std, prg, cnt
CONDITIONS
      CONSTANT RANGE:  -1095<=cnt<=4095
                    :  cmove has been elimidated.
```

Fig.5

○ : Supported

No	Optimization Function	MPA	MPB	SCH
1	Constant Folding	○	○	
2	Constant Propagation	○	○	
3	Copy Propagation	○	○	
4	Common Subexpression Elimination	○	○	
5	Dead Code Elimination	○	○	
6	Array/Subscript Optimization	○		
7	Invariant Expression Motion	○	○	
8	Strength Reduction	○		
9	Test Replacement	○		
10	Simple Store Elimination	○	○	
11	Parallel Induction Variable Elimination	○		
12	Localization of Subscript Computation	○		
13	Replacing Array-element with simple Variable	○		
14	User Function Inline Expansion	○		
15	Tail Recursion	○		
16	Inner Loop Unrolling	○		
17	Struct/Union Member Optimization with Aliasing Analysis	○		
18	Global Register Allocation		○	
19	Leaf Routine Optimization		○	
20	Peephole Optimization	○	○	
21	Branch Optimization			
	1) Code Straightening	○	○	
	2) Block Reordering	○	○	
	4) Branch to Branch Removal	○	○	
22	Instruction Scheduling			○

Fig. 6(A)

```
—
ENTRY No.:  1
OPTIMIZATION NAME:  constant folding
APPLICABLE TYPE OF INTERMEDIATE REPRESENTATION:  LINDA
      OPERATOR:  * (Don't care.)
      OPERANDS:  cnt
INTERMEDIATE REPRESENTATION DEPENDENCIES:
      CONSTANT RANGE:  *
APPLICATION CONDITIONS:
      Times:  -
DEPENDENCIES:
  -
—
ENTRY No.:  4
OPTIMIZATION NAME:  CSE
APPLICABLE TYPE OF INTERMEDIATE REPRESENTATION:  LINDA
      OPERATOR:  *
      OPERANDS:  std, prg, var, cnt
INTERMEDIATE REPRESENTATION DEPENDENCIES:
      CONSTANT RANGE:  * (Don't care)
      BOUNDARY:  X
      ADDRESS:  *
APPLICATION CONDITIONS:
      Times:  -
DEPENDENCIES:
  -
—
ENTRY No.:  14
OPTIMIZATION NAME:  User Function Inline
FUNCTION NAME:
APPLICABLE TYPE OF INTERMEDIATE REPRESENTATION:  LINDA
      OPERATOR:  *
      OPERANDS:  *
INTERMEDIATE REPRESENTATION DEPENDENCIES
APPLICATION CONDITIONS:
      Times:  1
DEPENDENCIES:
      Before 1, 2, ..., 13, 15, ..., 22
```

Fig. 6 (B)

```
--
ENTRY No.:  16
OPTIMIZATION NAME:  Inner Loop Unrolling
APPLICABLE TYPE OF INTERMEDIATE REPRESENTATION:  LINDA
      OPERATOR:  *
      OPERANDS:  *
INTERMEDIATE REPRESENTATION DEPENDENCIES:
APPLICATION CONDITIONS:
      Times:  1
DEPENDENCIES:
      After: 14, 1, 2, ...
--
ENTRY No.:  18
OPTIMIZATION NAME:  Global register allocation
APPLICABLE TYPE OF INTERMEDIATE REPRESENTATION:  LINDA
      OPERATOR:  *
      OPERANDS:  *
INTERMEDIATE REPRESENTATION DEPENDENCIES:
      CONSTANT RANGE:  * (Don't care)
      BOUNDARY:  X
      ADDRESS:  2
APPLICATION CONDITIONS:
      Times:  1
DEPENDENCIES:
      Before:  22
      After:  1, 2, 3, 4, ..., 17, 19, ..., 21
--
ENTRY No.:  22
OPTIMIZATION NAME:  Instruction Scheduling
APPLICABLE TYPE OF INTERMEDIATE REPRESENTATION:  LINDA
      OPERATOR:  *
      OPERANDS:  *
INTERMEDIATE REPRESENTATION DEPENDENCIES
      CONSTANT RANGE:  * (Don't care.)
      BOUNDARY:  bxd, axd
      ADDRESS:  2
APPLICATION CONDITIONS:
      Times:  1
DEPENDENCIES:
      Before:  6
      After:  1, 2, 3, 4, ...
```

Fig. 7(A)

Table of intermediate representations

```
struct SET_IML {
      IML_TYPE    Type of intermediate representation;
      CODE_TYPE   Operation code
      LIST        List of types
      LIST        List of operands (n)
      struct      Conditions {
            NUMBER address;
            RANGE const;
            FLAGS boundary:
      }
}
```

Fig. 7(B)

Data structure of table of optimization function

```
struct OPTIMIZE_TBL  {
      NUMBER      Optimization function number
      NAME        Optimization function name
      IML_TYPE    Type of intermediate representation
      LIST        List of applied operators
      LIST        List of applied operands
      struct cond_tag  {
            NUMBER address;
            RANGE const;
            FLAGS boundary;
                  :
      }
      struct opt_cond_tag  {
            NUMBER times;
      }
      List Before;
      List After;
}
```

Fig. 8

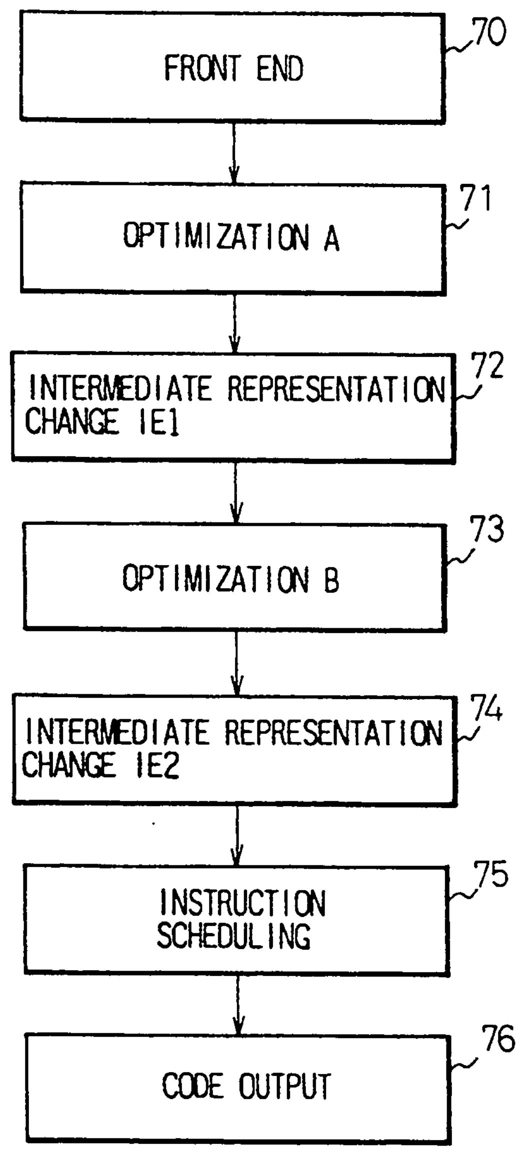

FRONT END
70
OPTIMIZATION A
71
INTERMEDIATE REPRESENTATION CHANGE IE1
72
OPTIMIZATION B
73
INTERMEDIATE REPRESENTATION CHANGE IE2
74
INSTRUCTION SCHEDULING
75
CODE OUTPUT
76

Fig.9

A SOURCE PROGRAM AND THE OPERATIONS OF A COMPILER ACCORDING TO AN EMBODIMENT OF THE PRESENT INVENTION

```
struct aaa {
  int  ii,jj;
  double fd;
} s_1,S_2;

double zz [100] , xx;

void sub(n,x)
int n;
double x;
{
  double         dd [100];
  int            i;

  s_1 = s_2;
  for (i = 0 ; i < n ; i++)
   {
     dd [i] = dd [i] + (xx+ x) = zz [i] ;
/*     s_1.ii = s_1.ii - s_1.jj; * /
    }
  subl(dd);
  return;
}
```

Fig.10

FRONT-END OUTPUTS OF THE EMBODIMENT

```
#label_1:
 cmove   axd:struct:("s 1") [0,0] axd("s 2") [0,0]

#label_2:
 move    var:("i") cnt:0
 bge     #label_9 var("i") var("n")

#label_6:
 mult    std#17d784 var("i") cnt:8
 load    std#17d7e8 axd("dd") [0, std:#17d784]
 add     std:#17d820 var:("xx") var:("x")
 mult    std:#17d858 var:("i") cnt:8
 load    std:#17d8bc axd("zz") [0  std:#17d858]
 mult    std:#17d8f4 std:#17d820 std:#17d8bc
 mult    std:#17d92c var:("i") cnt:8
 add     std:#17d990 std:#17d7e8 std:#17d8f4
 store   axd("dd")  [0, std:#17d92c] std:#17d990
 load    std:#17da74 axd("s_1") [0]
 load    std:#17dad8 axd("s_1") [4]
 sub     std:#17db3c std:#17da74: std:#17dad8
 store   axd("s_1") [0] std:#17db3c:(i4)

#label_7:
 move    std:#17dc80 var:("i")
 add     var:("i") std:#17dc80 cnt:1
 bit     #label_6  var:("i") var:("n")

#label_9:
 raturn
```

Fig.11

OUTPUTS OF OPTIMIZATION A OF THE EMBODIMENT

```
#label_1:
 cmove    axd:struct:("s_1") [0.0] axd ("s_2")
                       [0.0]

#label_2:
 move     prg:#17cddc("i")    cnt:0
 add      std:#183f6c cnt:1 var:("n")
 sub      prg:#1849b0 std:#183f6c cnt:1
 bge      #label 9 cnt:0 var:("n")

#label_10:
 add      prg:#17d818 var("xx") var:("x")
 move     prg:#184354 cnt:0
 shftrl   prg:#183f54 prg:#1849b0 cnt:1
 and      std:#1849e8 prg:#1849b0 cnt:1
 beq      #label_3  std:#1849e8 cnt:0

#label_11:
 load     std:#184b00 axd("dd") [0  prg:#184354]
 load     std:#184b84 axd("zz") [0  prg:#184354]
 mult     std:#184be0 prg:#17d818 std:#184b84
 add      std:#184c40 std:#184b00 std:#184be0
 store    axd:("dd") [0  prg:#184354] std:#184c40

 load     std:#184d7c axd("s_1") [0]
 load     std:#184e00 axd("s_1") [4]
 sub      std:#184e5c std:#184d7c std:#184e00
 store    axd:struct:("s_1") [0] std:#184e5c

 add      prg:#184354 prg:#184354 cnt:8
 beq      #label_9  prg:#183f5c cnt:0

; #label_3:
; #label_6:
 load     std:#17d7e0 axd("dd") [0  prg:#184354]
 load     std:#17d8b4 axd("zz") [0  prg:#184354]
 mult     std:#17d8ec prg:#17d818 std:#17d8b4
 add      std:#17d988 std:#17d7e0 std:#17d8ec
 store    axd("dd") [0  prg:#184354] std:#17d988

 load     std:#17da6c axd("s_1") [0]
 load     std:#17dad0 axd("s_1") [4]
 sub      std:#17db34 std:#17da6c std:#17dad0
 store    axd:("s_1")[0] std:#17db34

 load     std:#185044 axd("dd") [8  prg:#184354]
 load     std:#1850a4 axd("zz") [8  prg:#184354]
 mult     std:#185100 prg:#17d818 std:#1850a4
 add      std:#185160 std:#185044 std:#185100
 store    axd("dd") [8  prg:#184354] std:#185160

 load     std:#185278 axd("s_1") [0]
 load     std:#1852fc axd("s_1") [4]
 sub      std:#185358 std:#185278 std:#1852fc
 store    axd:("s_1")[0] std:#185358

; #label_7;
sid      none  sib:#17d9e8:line 22 (0)
 add      prg:#184354 prg:#184354 cnt:#185404:i4:16
 gub      srg:#183f5c prg:#183f5c cnt:#1854bc:i4:1
 bgt      #label_6  prg:#183f5c cnt:#185460:i4:0

; #label_9;
 raturn   none
```

Fig.12

INTERMEDIATE REPRESENTATIONS AFTER INTERMEDIATE REPRESENTATION CHANGE (IE1) OF THE EMBODIMENT (PART 1)

```
#label_0:
 entry    prg:#18fe28 prg:#18feac prg:#18ff5c
 store    bxd:("n")  [%fp+68] prg:#18fe28
 store    bxd:(ted:#1704b8)  [%fp+72] prg:#18feac
 store    bxd:(ted:#1704b8 [%fp+76] prg:#18ff5c

#label_1:
 movehi std:#18fffc adc:var:("s_1")+0
 or     std:#18fa8c std:#18fffc adc:("s_1")+0
 movehi std:#1900b4 abc:("s_2")+0
 or     std:#18fae4 std:#1900b4 adc:("s_2")+0
 add    prg:#19016c std:#18fa8c cnt:0
 add    prg:#190190 std:#18fae4 cnt:0
 move   prg:#1901dc cnt:16

#label_14:
 sub    prg:1901dc prg:#1901dc cnt:1
 load   std:#1904a0 bxd:("s_2")  [prg:#190190] +0 prg:#1901dc]
 store  bxd:("s_1")  [prg:#19016c] +0 prg:#1901dc] std:#1904a0
 cmp    prg:#18fdc8 prg:#1901dc cnt:0
 bne    #label_14(00190304) prg:#18fdc8

#label_2:
 load   std:#190e8c bxd:("n") [%fp+68]
 add    prg:#183e3c std:#190e8c cnt:1
 sub    prg:#1847d4 std:#183e3c cnt:1
 load   std:#190f2c bxd:("n") [%fp+68]
 cmp    prg:#1904e8 std:#190f2c cnt:0
 ble    #label_9 prg:#1904e8

#label_10:
 movehi std:#191000 adc:("XX")+0
 load   std:#190ff0 bxd:("XX") [std:#191000] +0 adc("XX"/9)+0]
 load   std:#1910e0 bxd:("X") [%fp+72]
 add    prg:#17d818 std:#190ff0 std:#1910e0
 move   prg:#184188 cnt:0
 srl    prg:#183e2c prg:#1847d4 cnt:1
 and    std:#18480c prg:#1847d4 cnt:1
 cmp    prg:#1904e8 std:#18480c cnt:0
 beq    #label_3 prg:#1904e8
```

Fig.13

INTERMEDIATE REPRESENTATIONS AFTER INTERMEDIATE REPRESENTATION CHANGE (IE1) OF THE EMBODIMENT (PART 2)

```
#label_11:
 add      std:#1902e4 var:(%fp) prg:#184188
 load     std:#184924 bxd:("dd") [std:#1902e4] +-800]
 movehi   std:#19126c adc:("zz"/6)+0
 or       std:#190340 std:#19126c adc("zz")+0
 load     std:#1849a8 bxd:("zz") [std:#190340] +0 prg:#184188]
 mult     std:#184a04 prg:#17d818 std:#1849a8
 add      std:#184a64 std:#184924 std:#184a04
 store    bxd("dd") [std:#1902e4] +-800] std:#184a64

 add      prg:#184188 prg:#184188 cnt:#1701cc:i4:8
 cmp      prg:#1904e8 prg:#183e2c cnt:#1700fc:i4:8
 beq      #label_9 prg:#1904e8


#label_3:
 add      prg:190198 var(%fp) prg:#184188

#label_6:
 load     std:#17d7e0 bxd("dd") [prg:#190198] +-800]
 movehi   std:#191480 adc("zz")+0
 or       std:#18fd80 std:#191480 adc("zz")+0
 load     std:#17d8b4 bxd:("zz") [std:#18fd80] +0 prg:#184188]
 mult     std:#17d8ec prg:#17d818 std:#17d8b4
 add      std:#17d988 std:#17d7e0 std:#17d8ec
 store    bxd:("dd")  [prg:#190198] +-800] std:#17d988
 load     std:#184c4c bxd("dd") [prg:#190198] +-792]
 movehi   std:#191620 adc("zz")+0
 or       std:#18fe98 std:#191620 adc("zz")+0
 add      std:#18fef4 std:#18fe98 prg:#184188
 load     std:#184cac bxd("zz") [std:#18fef4] +8]
 mult     std:#184d08 prg:#17d818 std:#184cac
 add      std:#184d68 std:#184c4c std:#184d08
 store    bxd("dd") [prg:#190198] +-792] std:#184d68

#label_7:
 add      prg:#184188 prg:#184188 cnt:16
 add      prg:#190198 prg:#190198 cnt:16
 sub      prg:#183e2c prg:#183e2c cnt:1
 cmp      prg:#1904e8 prg:#183e2c cnt:0
 bgt      #label 6 prg:#1904e8

#label_9:(0017dc04) /epi/term
 return   none
```

Fig. 14

OUTPUTS AFTER OPTIMIZATION B OF THE EMBODIMENT (PART 1)

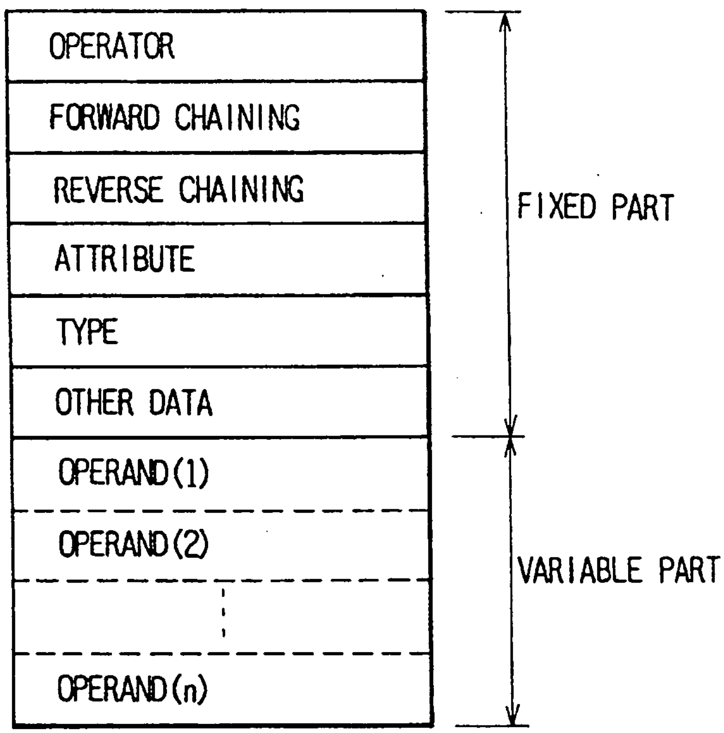
```
#label_0:
 entry    prg:#18fe28 prg:#18feac prg:#18ff5c
 store    bxd:("n")  [%fp+68] prg:#18fe28
 store    bxd:(ted:#1704b8)  [%fp+72] prg:#18feac
 store    bxd:(ted:#1704b8)[%fp+76] prg:#18ff5c

#label_1:
 movehi std:#18fffc adc:var:("s_1")+0
 or     std:#18fa8c std:#18fffc adc:("s_1")+0
 movehi std:#1900b4 abc:("s_2")+0
 or     std:#18fae4 std:#1900b4 adc:("s_2")+0
 add    prg:#19016c std:#18fa8c cnt:0
 add    prg:#190190 std:#18fae4 cnt:0
 move   prg:#1901dc cnt:16

#label_14:
 sub    (prg:#190904 prg:#1904f0) prg:#190904 cnt:#17ce38:u4:1
 load   std:#190bc8 bxd:("s_2")  [prg:#1908b8] +0 prg:#190904]
 store  bxd:("s_1") [prg:#190894] +0 prg:#190904] std:#190bc8
 bne    #label 14  prg:#1904f0

#label_2:
 add     std:#183e44  prg:#190550 cnt:1
 sub     prg:#1847dc std:#183e44 cnt:1
 cmp     prg:#1904f0 prg:#190550 cnt:0
 ble     #label_9  prg:#1904f0

#label_10:
 movehi  std:#191003 adc:("XX")+0
 load    prg:#1924c0 bxd:("XX")  [std:#191008] +0 adc("XX")+0]
 load    std:#1910e8 bxd:("X") [%fp+72]
 add     prg:#17d820 prg:#1924c0 std:#1910e8
 move    prg:#184190 cnt:0
 srl     prg:#183e34 prg:#1847dc cnt:1
 and     (std:#184314,prg:#1904f0) prg:#1847dc cnt:#170130:14:1
 beq     #:label_3 prg:#1904f0

#label_11(00184638)
 add     std:#1902ec var(%fp) prg:#184190
 load    std:#18492c bxd("dd") [std:#1902ec] +-800]
 movehi  std:#191274 adc:("zz")+0
 or      std:#190348 std:#191274 adc("zz")+0
 load    std:#1849b0 bxd("zz") [std:#190348] +0 prg:#184190]
 mult    std:#184a0c prg:#17d820 std:#1849b0
 add     std:#184a6c std:#18492c std:#184a0c
 store   bxd("dd") [std:#1902ec] +-800 std:#184a6c

 add     prg:#184190  prg:#184190 cnt:8
 cmp     prg:#1904f0 prg:#183e34 cnt:0
 beq     #label_9(0017d39c) prg:#1904f0
```

Fig.15

OUTPUTS AFTER OPTIMIZATION B OF THE EMBODIMENT (PART 2)

```
#label_3(0017d2e8)
 add     prg:#1901a0 var(%fp) prg:#184190
 movehi  std:#191488 adc("zz")+0
 or      prg:#18fd88 std:#191488 adc("zz")+0

#label_6(0017d6a0)  /entr
 load    std:#17d7e8 bxd("dd") [prg:#1901a0] +-800]
 load    std:#17d8bc bxd("zz") [prg:#18fd88] +0 prg:#184190]
 mult    std:#17d8f4 prg:#17d820 std:#17d8bc
 add     std:#17d990 std:#17d7e8 std:#17d8f4
 store   bxd("dd") [prg:#1901a0] +-800 ] std:#17d990

 load    std:#184c54 bxd("dd") [prg:#1901a0] +-792 ]
 add     std:#18fefc prg:#18fd88 prg:#184190
 load    std:#184cb4 bxd("zz") [std:#18fefc] +8 ]
 mult    std:#184d10 prg:#17d820 std:#184cb4
 add     std:#184d70 std:#184c54 std:#184d10
 store   bxd("dd") [prg:#1901a0] +-792 ] std:#184d70

#label_7:
 add     prg:#184190 prg:#184190 cnt:16
 add     prg:#1901a0 prg:#1901a0 cnt:16
 sub     (prg:#183e34,prg:#1904f0) prg:#183e34 cnt:1
 bgt     #albel_6 prg:#1904f0

#label_9:
 return  none
```

Fig.16


OPERATOR
FORWARD CHAINING
REVERSE CHAINING
ATTRIBUTE
TYPE
OTHER DATA
OPERAND(1)
OPERAND(2)
OPERAND(n)
FIXED PART
VARIABLE PART

COMPILING APPARATUS AND A COMPILING METHOD

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a compiling apparatus for improving the performance of and optimizing compiler.

Recent computers are required to operate at high speed. To improve the operation speed, processors that simultaneously execute a plurality of instructions and machines that carry out vector operations have been developed. These situations have raised the needs of optimizing compilers.

Compilers translate a source program written in a programming language such as C, FORTRAN, and COBOL into another language such as assembler. The "optimization" of compiling means to translate a source program in such a way as to improve the execution speed of a translated program and reduce a memory area required by the translated program, without changing the meaning of the source program.

Generally, compilers optimize a source program through intermediate representations, which are synonymous with intermediate languages, intermediate texts, or internal representations. The intermediate representations are independent of a computer or a work station that executes a translated program. Some of the intermediate representations are eliminated, moved, and integrated through optimization, to thereby provide optimized object codes. The selection of intermediate representations, therefore, affects the performance and speed of optimization. If the types and contents of the intermediate representations are fixed according to a specific stand point, the optimization will frequently be unsuccessful because there are various kinds of optimization and final objects that will not fit the fixed intermediate representations.

The present invention relates to optimizing a compiler that translates a programming language into an assembler language or a machine language.

2. Description of the Related Art

Optimization means deleting, moving, and integrating intermediate representations, to provide high-speed small-sized efficient object code.

The intermediate representations must be independent of languages and machines when achieving the optimization and providing common code generators and standardized parts. The intermediate representations are explained in, for example, a reference 1 "Compiler: principles, techniques, and tools" by A.V. Aho et al, Science, and a reference 2 "A way of thinking of intermediate languages," Interface, March 1989, pp 211 to 219.

Conventional optimization involves a front end, an optimizing section, and a code output section.

The front end provides intermediate representations according to a source program. The intermediate representations are optimized by the optimizing section, to improve the execution speed of the program and reduce the size of the program. The optimized result is supplied to the code output section, which provides code based on a machine-dependent programming language such as a machine language dedicated to a target machine.

The intermediate representations optimized by the optimizing section are independent of languages and machines.

There are compilers that change the structure of intermediate representations during optimization. Such compilers are explained in a reference 3 "Optimizing compilers for SPARC" by S. S. Muchnick, Sun Technology, 1988, and in a reference 4 "A global optimizer for Sun FORTRAN, C and Pascal" by Ghodssi, Vida, S. S. Muchnick, and Alex Wu, USENIX Conference, 1986. These compilers mainly substitute existing processes for partial processes during compilation and change the physical memory map of data structure of intermediate representations.

Japanese Unexamined Patent Publication No. 2-81137 discloses the structure of an optimizing compiler. Objects of this disclosure are to (1) efficiently optimize a source program, (2) efficiently collect optimization data, (3) quickly find optimization functions to achieve, (4) eliminate side effect elements of the source program, and (5) clear side effects of the source program. This prior art initially changes an intermediate language (intermediate representations) and optimizes the changed one.

As explained above, the conventional compilers design and prepare an optimizing section according to fixed intermediate representations, causing the following problems:

(1) Machine instructions and intermediate representations do not always have a one-to-one relationship. For example, if an instruction scheduling function that determines the positions of different machine instructions is compiled under a single intermediate representation, the intermediate representation will be unable to optimize the different machine instructions involved in the function.

(2) Even if all intermediate representations provided by a front end are related to machine instructions, respectively, there will still be the following problems:

(a) The intermediate representations provided by the front end must be independent of a target machine. If not so, different intermediate representations must be prepared for different target machines, and the front end must be renewed whenever a compiler is prepared.

(b) Relating instructions and intermediate representations to each other in a one-to-one relationship in the front end increases the number of the intermediate representations, to thereby complicate optimization and elongate an optimization time. For example, if a target machine is designed for 32-bit instructions and if it is impossible to load or store a 32-bit address at one time, two instructions are combined to substitute for a load/store instruction to address 32 bits. If the front end relates the two instructions to different intermediate representations, it is necessary to recognize during optimization that the two specific instructions serve as the single load/store instruction. This is not as simple as recognizing a single load or store instruction and complicates the optimization.

(3) Each optimization function is applied only to a predetermined intermediate representation. This puts limits on the front end and on optimization.

The intermediate representation changing techniques disclosed in the references 3 and 4 completely change

the physical data structures of intermediate representations. Accordingly, these techniques hardly standardize optimization functions.

SUMMARY OF THE INVENTION

An object of the present invention is to provide a compiling apparatus and a compiling method that meet a variety of requirements for optimization. A compiler according to the present invention provides a high-performance object code according to a target architecture, a source program, and optimization requirements. The present invention standardizes intermediate representation changing processes and optimizing processes, to provide an efficient compiler.

In order to accomplish the objects, the present invention provides a compiling apparatus having a front end for providing intermediate representations according to a source program, an optimizing unit for optimizing the intermediate representations, at least one intermediate representation changing unit for changing the optimized intermediate representations, and a code output unit for providing codes according to the last obtained intermediate representations. The compiling apparatus is characterized by an optimizing structure determination unit for determining the number of repetitions of an optimization phase achieved by the optimizing unit and intermediate representation changing unit and selecting optimization functions carried out in each of the optimization phases. According to the determination by the optimizing structure determination unit, the intermediate representations are changed and optimized in each of the optimization phases.

The optimizing structure determination unit determines the number of the intermediate representation changing and optimizing processes according to the program-dependent intermediate representations provided by the front end, object-architecture-dependent intermediate representations, and the selected optimization functions.

The physical memory map structure of main data of the intermediate representations during the intermediate representation changing and optimizing processes is fixed.

The fixed physical memory map structure helps standardize the intermediate representation changing and optimizing processes.

The optimizing structure determination unit may select a plurality of optimizing structures, separately carry out optimization according to each of the selected structures, and select the most proper one among the optimizing structures according to the optimization results.

The optimizing unit and intermediate representation changing unit may repeat predetermined optimization.

The compiling apparatus may have a switch that connects and disconnects the optimizing structure determination unit in front of the optimizing unit and intermediate representation changing unit.

When the switch is turned ON, the optimizing structure determination unit determines an optimizing structure in each compiling phase. According to the optimizing structure, intermediate representations corresponding to a source program are changed, and optimization function to be applied are selected.

When the switch is turned OFF, the optimizing unit and intermediate representation changing unit repeat predetermined intermediate representation optimizing and changing processes a predetermined number of times.

The optimizing unit and intermediate representation changing unit may have a variable identifying a present phase when repeating the same optimization.

The compiling apparatus may have a selection switch to determine whether or not an optional one of the optimization functions must be achieved in an optional one of the optimizing phases.

The selection switch may be externally controlled to stop an optional optimization function in an optional phase.

The compiling apparatus may have a printing unit for printing intermediate representations in each phase and a selection switch for selecting the printing unit.

The selection switch may be externally controlled to print intermediate representations at optional timing in any one of the phases.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be more clearly understood from the description as set forth below with reference to the accompanying drawings, in which:

FIG. 1 explains a conventional optimizing compiler;

FIG. 2 shows a principle of the present invention;

FIG. 3 explains an optimizing structure determination flow;

FIG. 4(A) shows examples of intermediate representations at an input end of optimization;

FIG. 4(B) shows examples of intermediate representations at an output end of the optimization;

FIG. 5 shows optimization functions and their classes;

FIG. 6(A) shows examples of optimization data stored in a table;

FIG. 6(B) shows other examples of optimization data stored in a table;

FIG. 7(A) shows a data structure in a table used for changing intermediate representations;

FIG. 7(B) shows a data structure in a table used for optimization;

FIG. 8 shows a compiler according to an embodiment of the present invention;

FIG. 9 shows an example of a program used for explaining the operation of the compiler according to the embodiment of the present invention;

FIG. 10 shows outputs of a front end with respect to the program, according to the embodiment of the present invention;

FIG. 11 shows outputs after optimization A according to the embodiment;

FIG. 12 shows intermediate representations after a change (IE1) according to the embodiment;

FIG. 13 shows other intermediate representations after the change (IE1);

FIG. 14 shows outputs after optimization B according to the embodiment;

FIG. 15 shows other outputs after the optimization B; and

FIG. 16 explains the data structure of an intermediate representation.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

Before describing the preferred embodiments according to the present invention, an example of the related art is provided.

FIG. 1 explains optimization according to the prior art. A front end **100** provides intermediate representations according to a source program, so that the source program can be compiled. An optimizing section **101** optimizes the intermediate representations provided by the front end **100**, to improve the speed of the source program and reduce the size of the source program. A result of the optimization is supplied to a code output section **102**, which provides code in a machine-dependent program language such as a machine language.

The intermediate representations optimized by the optimizing section **101** are independent of languages and machines.

FIG. 2 shows a principle of the present invention.

In the figure, numeral **1** is a source program, **2** is a front end for providing intermediate representations corresponding to the source program **1**, **3** is a switch that is usually turned ON to determine an optimizing structure, **4** is an optimizing structure determination unit, **5** is an optimizing unit, **6** is an intermediate representation changing unit, **7** is a determination unit, **8** is a code output unit, and **9** is an object program. The optimizing structure determination unit **4** may be arranged inside or outside a compiler. Determined results of the unit **4** are provided to the optimizing unit **5** and intermediate representation changing unit **6**.

An intermediate representation data file **10** stores intermediate representation data used for providing first intermediate representations from the front end **2**, or for providing an object program. An optimization data file **11** stores optimization data used for determining an optimizing structure. The file **11** also stores determined optimization data. Intermediate representation structure data **12** is provided by the front end **2**. Intermediate representation structure data **13** is provided by the intermediate representation changing unit **6**.

According to the source program **1** provided to the front end **2** and according to the target machine, the optimizing structure determination unit **4** selects optimization functions to achieve and determine the number of repetitions of an intermediate representation changing process. According to the determination, a loop operation of the optimizing unit **5** and intermediate representation changing unit **6** is repeated. The same optimization may be repeated several times, to provide a high-performance object program corresponding to the source program and target machine.

In FIG. 2, the front end **2** changes the source program **1** into the intermediate representations. The switch **3** determines whether or not the optimizing structure determination unit **4** is used. The switch **3** may be optionally turned ON and OFF according to required optimization levels.

When the switch **3** is OFF, the optimizing unit **5** and intermediate representation changing unit **6** repeat a predetermined optimizing phase involving optimization and intermediate representation change a predetermined number of times. The optimizing phase is prepared according to the source program **1**. When the determination unit **7** counts the predetermined number of times, the code output unit **8** prepares codes according to the last optimized and changed intermediate representations. These codes form the object program **9**.

When the switch **3** is ON, the optimizing structure determination unit **4** arranged inside or outside the compiler obtains preliminary supplied front end intermediate representation data and target machine data from the intermediate representation data file **10**. The unit **4** also obtains optimization data such as required optimization functions from the optimization data file **11**. Then, the unit **4** determines the number of intermediate representation optimizing and changing processes. For example, for a simple optimizing structure, the unit **4** may determine one time of optimizing and changing source-program-dependent intermediate representations, and one time of optimizing and changing target-machine-dependent intermediate representations.

Once the optimizing structure determination unit **4** selects the optimization functions to achieve and the number of intermediate representation optimizing and changing processes, these results are informed to the optimizing unit **5**, intermediate representation changing unit **6**, and determination unit **7**. At first, the optimizing unit **5** optimizes the source-program-dependent intermediate representations according to the intermediate representation structure data **12** provided by the front end **2**. Thereafter, the intermediate representation changing unit **6** changes a part or the whole of the optimized intermediate representations, to provide the intermediate representation structure data **13**. The determination unit **7** repeats a loop of the intermediate representation optimizing and changing processes until the number determined by the optimizing structure determination unit **4** is counted. In each loop, the optimizing unit **5** carries out the specified optimization functions, and the intermediate representation changing unit **6** changes the optimized intermediate representations.

When the determination unit **7** counts the predetermined number, the code output unit **8** provides codes to form the object program **9** in machine language according to the last provided intermediate representation structure data **13**.

Each of the optimizing unit **5** and intermediate representation changing unit **6** may have a function of determining whether or not the predetermined number of the intermediate representation optimizing and changing processes have been completed. In this case, the determination unit **7** may be omitted.

The optimizing structure determination unit **4** is able to repeatedly change and optimize intermediate representations, so that optimization functions that have not been applied to first intermediate representations may be applied to secondly changed intermediate representations. Instruction scheduling may be carried out and optimized in a stage where machine instructions and intermediate representations have a one-to-one relationship. This results in improving the optimizing performance of the compiler.

The optimizing unit **5** may have an intermediate representation changing function, to omit the intermediate representation changing unit **6**.

FIG. 3 shows the flow of steps for determining an optimizing structure and related data.

Data **20** corresponds to the data **10** of FIG. 1 and indicates the characteristics of source-program-dependent intermediate representations provided by the front end. Data **21** indicates the characteristics of target-machine-dependent intermediate representations supplied to the code output unit that provides object codes after optimization. Optimization data **22** corresponds to the data **11** of FIG. 1 and is used to determine optimization functions and the number of repetitions of intermediate representation optimizing and changing processes.

In FIG. 3, step S1 picks up optimization functions to be held in a compiler. The optimization data **22** includes a list of optimization functions corresponding to the

front-end intermediate representation data **20** and code-providing intermediate representation data **21**, and the step **S1** selects some of the optimization functions from the data **22**, so that they are executed by the optimizing compiler. This selection may be made by a person who prepares the compiler.

Step **S2** classifies the selected optimization functions into execution classes. At this time, the optimization functions are classified into those that are achieved on the front-end intermediate representations, those that are achieved on the code-providing intermediate representations, those that are achieved between the above two cases, and those that are achieved in instruction scheduling. The optimization functions are classified according to:

(1) the physical memory map structures of data of intermediate representations to be optimized, and

(2) objects to be optimized.

The item (2) means that a given optimization function is classified according to whether the function is carried out on a language or program structure such as a loop and a conditional statement, on a target structure such as a parallel structure and a register structure, or on both of them. The classified result is stored in the optimization data **22**.

Step **S3** determines a compiler structure in two steps **S30** and **S31**. The step **S30** determines the number of times of changing intermediate representations. In principle, this number corresponds to the number of classes of the classified optimization functions. The number may also be determined according to the types of intermediate representations and the intermediate representation dependencies of the optimization functions.

The step **S31** determines the optimization functions in three steps **S310**, **S311**, and **S312**. The step **S310** sorts the optimization functions on execution phases with reference to the types of intermediate representations and the intermediate representation dependencies. The step **S311** extracts applicable ones among the optimization functions according to the application conditions of the functions. The step **S312** determines the execution order of the optimization functions according to the front-end and code-providing dependencies of the functions.

Once the optimizing structure is determined, the intermediate representation optimizing and changing processes are carried out according to the determined results. Namely, the intermediate representations are changed and optimized phase by phase, to provide codes that form an optimized object program.

The processes of FIG. **3** for determining an optimizing structure will be explained in detail with reference to FIGS. **4**(A) to **7**(B).

This explanation relates to a compiler for a parallel RISC (reduced instruction set computer). **10** According to the RISC architecture, a memory is accessed only with a load/store instruction, and operations are always carried out between registers, to improve hardware speed. The parallel RISC architecture has a function of simultaneously executing a plurality of instructions. Although the following explanation relates to the compiler for the parallel RISC architecture, the present invention is applicable for any other processors. In the following explanation, a front end is prepared for a language, with no regard to targets. This provision is right because the specifications of a language are unchanged for different targets.

FIGS. **4**(A) and **4**(B) show examples of intermediate representations at I/O sections of a compiler, in which FIG. **4**(A) shows intermediate representations provided by a front end, and FIG. **4**(B) shows intermediate representations for providing output codes. The intermediate representations of FIG. **4**(A) are dependent on a source program, and those of FIG. **4**(B) are dependent on a target architecture. These intermediate representations correspond to the data **20** and **21** of FIG. **3** used to determine an optimizing structure.

FIG. **4**(A) shows the attributes of each intermediate representation corresponding to a code (an operator). The attributes include the type, code, data types, operands, and conditions of the intermediate representation.

Taking the first intermediate representation of Fig. **4**(A) as an example, the type is a generic LINDA, the code is an operator LOAD, the data types are I1 to I4 representing signed data of one to four bytes, **u1** to **u4** representing unsigned data of one to four bytes, **r8** to **r16** representing floating-point data of 8 to 16 bytes, and **c8** to **c32** representing fixed-point data of 8 to 32 bytes, the operands are one or two, and the conditions are a boundary (an unguaranteed boundary X) and an address representation of three values. Similarly, intermediate representations for codes (operators) ADD and cmove are shown in FIG. **4**(A).

FIG. **4**(B) shows the intermediate representations for providing codes that are dependent on the target architecture. Although the intermediate representations of FIG. **4**(B) resemble those of FIG. **4**(A), the contents of them do not always agree with each other. For example, the data types of the code LOAD of FIG. **4**(B) do not include **r16** and **c8** to **c32**, and the intermediate representation for the code "cmove" is not included in FIG. **4**(B).

FIG. **5** shows optimization functions to be achieved by the compiler. These functions are picked up in the step **S1** of FIG. **3**. In FIG. **5**, each of the optimization functions numbered from 1 to 22 has a title. For example, the optimization function No. 1 is titled "CONSTANT FOLDING" to fold constants. This function folds, for example, "1+3" as "4." The optimization function No. 2 is titled "CONSTANT PROPAGATION" to propagate constants. Reference marks MPA, MPB, and SCH shown in FIG. **5** will be explained later. These data are in the form of a table or a database and are stored as the optimization data **22** of FIG. **3**.

FIGS. **6**(A) and **6**(B) show examples of the optimization functions stored in the table. The figures show only data for entry numbers **1**, **4**, **14**, **16**, **18**, and **22**. These entry numbers correspond to the optimization function numbers shown in FIG. **5**. Taking the entry number **1** as an example, the title is "constant folding", the type of applicable intermediate representation is "LINDA", an operator is "*" (don't care) to mean that this optimization function is irrelevant to operators, an operand is "cnt" (a constant), and intermediate representation dependencies are shown.

The entry number **4** has the title "CSE" (common subexpression elimination) and, as an intermediate representation dependency, a boundary of X to indicate unguaranteed. In this way, each of the optimization functions corresponding to the entry numbers **14**, **16**, **18**, and **22** has data shown in FIGS. **6**(A) and **6**(B).

According to the data shown in FIGS. **4**(A), **4**(B), **6**(A), and **6**(B), the optimizing structure determining flow of FIG. **3** is carried out.

If the intermediate representations provided by the front end and the intermediate representations for providing codes completely agree with each other, it is not necessary to change the intermediate representations. If they do not agree with each other as shown in FIGS. 4(A) and 4(B), the optimizing structure determining flow of FIG. 3 is carried out.

Namely, the step S1 of FIG. 3 picks up optimization functions as shown in FIG. 5. The step S2 of FIG. 3 classifies the optimization functions according to the conditions (1) and (2) explained above. Since this embodiment employs the same physical memory map structure for data of each intermediate representation, it is not necessary to classify the optimization functions according to the condition (1). According to the condition (2), the language- or program structure-dependent optimization functions are achieved on a source program structure or on variables, and the architecture-dependent optimization functions are achieved on the specifications of a target machine architecture.

Memory addressing instructions such as a register allocation instruction and an instruction scheduling instruction are dependent on architecture. Accordingly, optimization functions carried out on these instructions are also the architecture-dependent optimization functions. According to the embodiment of the present invention, the architecture-dependent optimization functions are classified into two as follows:

(1) optimization functions that manipulate instructions themselves, e.g., reducing the number of instructions and changing an instruction into another of higher speed (for example, changing a multiplication instruction into a repetition of addition)

(2) optimization functions related to the order of instructions

The optimization functions 1 to 22 of FIG. 5 are classified according to these criteria into classes MPA, MPB, and SCH. Here, the class MPA includes language-dependent (source-program-dependent) optimization functions to be carried out on language-dependent intermediate representations, the class MPB includes architecture-dependent (target-machine-dependent) optimization functions to be carried out on architecture-dependent intermediate representations, and the class SCH includes optimization functions related to the execution order of instructions among the architecture-dependent optimization functions.

In FIG. 5, any optimization function with a plurality of marks o is repeatedly carried out on different kinds of intermediate representations.

After the classification, the step S3 of FIG. 3 determines a compiling structure. The step S30 determines the number of times of changing intermediate representations according to the types of intermediate representations and the number of intermediate representation dependencies shown in FIGS. 6(A) and 6(B).

In this embodiment, the intermediate representations are of only one type, LINDA. The number of intermediate representation dependencies is two for the "boundary" excluding those with a mark "*" (don't care). The two intermediate representation dependencies are as follows:

boundary: bxd, axd address 2
(referred to as phase 3)

boundary: x address 2
(referred to as phase 2)

In addition, there is the following output from the front end:

boundary: x address 3
(referred to as phase 1)

Accordingly, the intermediate representations must be changed three times.

The step S31 of FIG. 3 sets the timing and positions to carry out the optimization functions.

According to the types of the intermediate representations and the intermediate representation dependencies of the optimization functions shown in Figs. 6(A) and 6(B), the step S310 of FIG. 3 sorts the optimization functions on the phases 1, 2, and 3. The entry numbers (function numbers) 1, 4, 14, and so on of FIGS. 6(A) and 6(B) are sorted as follows:

Phase 1: 1, 4, 14, 16
Phase 2: 1, 4, 14, 16, 18
Phase 3: 22

The step 311 of FIG. 3 extracts proper ones among these optimization functions according to application conditions as follows:

Phase 1: 1, 4, 14, 16
Phase 2: 1, 4, 18
Phase 3: 22

The step 312 of FIG. 3 determines the execution order of the optimization functions according to the dependencies as follows:

Phase 1: 14, (1, 4), 16
Phase 2: (1, 4), 18
Phase 3: 22 where (1, 4) means either of 1 or 4 may be carried out at first.

In this way, the optimizing structure is determined. According to the determination, the intermediate representations are automatically optimized and changed. For automatization, the following must be prepared:

1) A data structure each element of which corresponds to an item of a table
2) Data encoded into a string of characters or numbers to be held as an internal structure of the compiler

FIG. 7(A) shows a table used to change intermediate representations, and FIG. 7(B) shows a table used for carrying out an optimization function. In the figures, IML is an intermediate representation. The table of FIG. 7(A) for a given intermediate representation contains the type of the intermediate representation (IML-TYPE), an operation code (CODE-TYPE), a list of data types, a list of operands, etc. The table of FIG. 7(B) for a given optimization function contains an optimization function number, the name of the optimization function, the type of an intermediate representation on which the optimization function is carried out, a list of applied operators, etc. These data are stored in the optimization data file 11 of FIG. 2.

An embodiment of the present invention will be explained with reference to FIGS. 8 to 16.

FIG. 8 shows the structure of a compiler according to the embodiment. This structure is provided by the optimizing structure determining flow of FIG. 3.

The optimizing structure of a compiler for a program for the parallel RISC can be obtained as explained with reference to FIGS. 4(A) and 4(B) to 7(A) and 7(B). The structure of FIG. 8 is obtainable in the same manner.

The structure of FIG. 8 carries out optimization three times in phases 1 to 3, which correspond to the classes MPA, MPB, and SCH of FIG. 5. Intermediate representations are changed three times, accordingly. Among the three times of changing, the first one is carried out by a front end to change a source program into first intermediate representations.

In FIG. 8, the front end 70 provides the first intermediate representations according to the source program. Optimization A (71) optimizes the first intermediate representations provided by the front end 70. The optimization A involves the optimization functions classified into the class MPA (phase 1) of FIG. 5. Interface expansion IE1 (72) changes the optimized intermediate representations, to provide intermediate representations of target instruction level. Optimization B (73) optimizes these intermediate representations.

The optimization B involves the optimization functions classified into the class MPB (phase 2) of FIG. 5. Interface expansion IE2 (74) changes the optimized intermediate representations, to provide intermediate representations of instruction scheduling level. Instruction scheduling 75 is carried out on these intermediate representations. The instruction scheduling 75 involves the optimization function 22 of the class SCH (phase 3) of FIG. 5. According to a result of this optimization function, a code output section 76 provides codes.

According to this embodiment, the front end provides the first intermediate representations. Generally, these intermediate representations provided by the front end are irrelevant to a target. If not so, different intermediate representations must be prepared for different targets, and therefore, a new front end must be prepared whenever a compiler is formed.

When adding a new optimization function to the structure of FIG. 8, or when changing the specifications of optimization, the steps S1 to S3 of FIG. 3 may be repeated to determine a compiler structure. If the number of classes of optimization functions is increased, a step of changing intermediate representations will be added, to form a compiler structure that provides a high-performance object.

The optimizing structure determination flow of FIG. 3 may be carried out by externally selecting and setting required data. If the specifications of optimization functions are registered in tables as shown in FIGS. 7(A) and 7(B), proper ones of the functions may 10 be automatically picked up. In this case, the optimizing structure determination may be omitted. Namely, this case corresponds to turning OFF the switch 3 of FIG. 2.

It is possible to select a plurality of optimizing structures and separately operate them. According to the results of the operations, the best one of the optimizing structures will be selected to provide a high-performance object.

When operating the plurality of optimizing structures, the following conditions are applied:

(1) If there are a plurality of processors, they will be allocated for the optimizing structures, respectively.

(2) If there is one processor, the optimizing structures are operated one after another before providing codes. Lists of resultant intermediate representations of the respective optimizing structures are stored.

An execution time is estimated for each of the listed elements, i.e., for a string of the intermediate representations of each of the optimizing structures, and object codes are prepared according to the intermediate representations that provide the shortest execution time. The execution time is calculable because instructions corresponding to the intermediate representations are known, and the execution time of each of these instructions is known because the execution time is architectural data required for instruction scheduling. If there are loops and if the number of loops is statically known, the execution time of instructions will be multiplied by the number of loops. If the number of loops is unknown, the execution time may be multiplied by the same constant. In this case, no serious error will occur because every code is under the same condition.

The flow of FIG. 3 can be incorporated as a compiling phase in a compiler, to select optimization functions according to a source program.

Additional functions of the embodiment of FIG. 8 will be explained.

According to the embodiment of FIG. 8, intermediate representations provided by the front end are repeatedly optimized and changed through phases. A selection switch (different from the switch 3 of FIG. 2) may be provided for each of the phases, to determine whether or not the intermediate representation optimizing and changing processes of a corresponding phase must be carried out. The switches may be controlled from outside of the compiler, to stop the processes of an optional phase. This function may be provided as an option of the compiler. If the option stands, the corresponding phase is passed (returned). This reduces a translation time and improves the efficiency of debugging when preparing a compiler.

The arrangement of FIG. 8 may have a function of printing intermediate representations. Namely, a selection switch may be arranged to determine whether or not intermediate representations in each phase are printed. This selection switch is different from the selection switch for selecting whether or not the optimizing and changing processes of a given phase are carried out. The printing function of intermediate representations may be called by controlling an option during the debugging of a program or the processing of the compiler, to print intermediate representations at an optional location of an optional phase, to improve debugging efficiency.

FIGS. 9 to 16 show examples of intermediate representations at each optimization phase with respect to a program, according to the compiling structure of FIG. 8.

FIG. 9 shows a program for explaining the operation of the compiler according to the embodiment, FIG. 10 shows outputs of the front end with respect to the program, FIG. 11 shows outputs after the optimization A, FIGS. 12 and 13 are intermediate representations after the intermediate representation change IE1, and FIGS. 14 and 15 show outputs after the optimization B.

Essential ones of the marks shown in FIGS. 10 to 16 will be explained.

#label-n ($0<=n$) is a base block. Intermediate texts forming the block follow the label. Each of the intermediate texts includes an operator and a string of operands. The operator is separated from the operands by a space.

The operands are as follows:

var: a variable

std, prg: a temporary variable produced by the compiler

axd, bxd: an array, structure data

cnt: a constant

The front end provides intermediate representations shown in FIG. **10** with respect to the source program of FIG. **9**. In FIG. **10**, the intermediate representations and instructions do not always correspond to each other in a one-to-one relationship. This will result in insufficient optimization.

With respect to these outputs of the front end, the optimization A (**71** of FIG. **8**) is carried out. Namely, the optimization functions classified into the class MPA of FIG. **5** are achieved on the front-end outputs. FIG. **11** shows results of the optimization A. Comparing FIGS. **10** and **11** with each other, the effect of the optimization is apparent. Namely, a variable XX and array elements dd(i) are collectively handled. A structure assignment is achieved with a single intermediate representation.

The intermediate representations provided by the optimization A of FIG. **11** are changed by the interface expansion IE1 (**72** of FIG. **8**) into intermediate representations shown in FIGS. **12** and **13**. The interface expansion IE1 changes the intermediate representations provided by the optimization A according to target instruction sets and the optimization functions included in the next optimization B (**73** of FIG. **8**). An object of the interface expansion IE1 is to provide clear intermediate representations as an object of the optimization B.

As shown in FIGS. **12** and **13**, the XX and dd(i) have increased the number of intermediate representations and changed the operands according to addressing. A structure assignment statement has been changed into a loop structure. The intermediate representation change by the interface expansion IE1 is achievable according to branch tables because an output is uniquely determined once the code and operands of a corresponding input intermediate representation are determined. Other techniques are also employable.

With respect to the outputs of the interface expansion IE1 of FIGS. **12** and **13**, the optimization B (**73** of FIG. **8**) is carried out. Objects of the optimization B are target-dependent parts that do not directly appear in the source program, or parts that have not been analyzed in the front end. As shown in FIG. **5**, these parts overlap those for the optimization A. According to this embodiment, the following items are also objects of the optimization B:

1. Optimization of address calculations
2. Loop processes of structure assignment

According to this embodiment, the functions of the optimization B are basically the same as those of the optimization A. Accordingly, the optimization B is achieved by calling the same functions as those for the optimization A as much as possible. For this purpose, the types of the intermediate representations must be considered. The repetition of the optimization will be explained later.

The prior art does not carry out the optimization B of the present invention. Namely, the prior art does not achieve the optimization functions included in the optimization B, so that, if the RISC architecture is a target, a high-performance object will not be provided. Accordingly, the interface expansion IE1 (**72** of FIG. **8**) and optimization B are indispensable processes.

FIGS. **14** and **15** show results of the optimization B. Comparing the intermediate representations after the interface expansion IE1 of FIGS. **12** and **13** with the results of the optimization B of FIGS. **14** and **15**, it will be understood that basic optimization is achieved after the interface expansion IE1.

According to the embodiment of FIG. **8**, the functions of the optimization A and those of the optimization B are standardized. According to another embodiment, the optimization A and optimization B may be achieved with separate functions.

When changing the intermediate representations by the phase IE1 (**72** of FIG. **8**) of the embodiment of FIG. **8**, the instructions and intermediate representations are not completely in a one-to-one relationship. This is because processes after the phase IE1 are classified into the following two groups according to functions and data to use:

(1) The optimization B and register assignment

(2) Instruction scheduling

The item (1) above relates to providing instructions as follows:

(a) Deleting dead instructions

(b) Collecting overlapping instructions

(c) Providing speedier instructions

The item (2) above does not relate to providing instructions but improves hardware efficiency by properly setting the execution order of instructions. Accordingly, it is preferable not to put the instructions and intermediate representations in a one-to-one relationship just after the intermediate representation changing of the phase IE1.

In the instruction scheduling, a double-precision load/store instruction must be divided into two single-precision instructions. The reason is because these two instructions can be executed in parallel with each other. Namely, these instructions will be more quickly completed if they are separately executed than if they are sequentially executed. In terms of register assignment and optimization, however, the two single-precision instructions originally indicate a single entity. Accordingly, if they are separately handled, the operation of the compiler during the optimization A will be complicated. In addition, in terms of the register assignment, consecutive registers must be allocated for the two separate instructions, so that information indicating that the two instructions are continuous must be provided for the register assignment.

On the other hand, the intermediate representation changing phase IE2 (**74** of FIG. **8**) relates to the instruction scheduling, so that the intermediate representations and actual instructions are decomposed to form one-to-one relationship. This is achieved as in the case of the intermediate representation changing phase IE1 (**72** of FIG. **8**).

According to the embodiment, the instruction scheduling is carried out after the register assignment, according to a known technique. Instead of carrying out the instruction scheduling after the register assignment, the following approaches 1 to 4 may be employable:

1. Carrying out the instruction scheduling before the register assignment
2. Simultaneously carrying out the instruction scheduling and register assignment
3. Repeating the register assignment and instruction scheduling in optional order

4. Repeating the register assignment and instruction scheduling in optional order and carrying out optimization during the repetition.

In each of these cases, instructions and intermediate representations are related to each other in a one-to-one relationship before the instruction scheduling, to determine how to change the intermediate representations, i.e., to determine the timing of the instruction scheduling.

The types of intermediate representations will be explained. According to the embodiment, each intermediate representation has a data structure shown in FIG. 16. To make an actual program according to the structure of FIG. 16, the following structure declaration is made, so that the number of operands becomes variable depending on an operator.

struct OperandType operand[1];

In this way, the physical memory map structure of intermediate representations can be fixed between phases of a compiler.

Changing intermediate representations will be explained. In the following explanation, the "instructional intermediate representation" means an intermediate representation corresponding to an instruction. For example, #label-2 of FIG. 10 contains the following parts:

move var:("1") cnt: 0

bqe #label-o var:("1") var:("n") Here, "move var:("1") cnt:0" is an instructional intermediate representation. The "move" is an operator or an instructional intermediate representation code, the "var:("1")" is an operand 1 (first operand), and "cnt:0" is an operand 2 (second operand). Changing an intermediate representation means:

(1) Changing an operand pointed to by the intermediate representation. In FIGS. 9 to 15, examples of this case will be seen as changes in operands of instructional intermediate representations. This is carried out by shifting the pointer. In FIG. 10, changes in operands of an instructional intermediate representation are as follows:

"move var:("1") cont:0" of #label-2 in the front end output of FIG. 10 is changed to "move prg:#17cddc("1") cnt:0" of #label-2 in the output of the optimization A of FIG. 11.

(2) Changing the operator of the intermediate representation

(3) Changing the output method of the intermediate representation. This means to provide a plurality of instructional intermediate representations or a loop structure according to an instructional intermediate representation. For example, "cmove" in FIGS. 10 and 11 is changed into a loop of #label-14 after the phase IE1 as shown in FIG. 12 and after the optimization B as shown in FIG. 14.

In this way, the meanings of the intermediate representations are changeable without changing their data structures.

Repetition of optimization will now be explained.

This embodiment standardizes intermediate representations to be optimized, to carry out uniform optimization in each optimization phase. A variable indicating the presently processed phase is used, and individual optimizing routines are standardized.

The optimization can be repeatedly called in the same phase. The repetition of the optimization is achievable in the following methods:

(1) A person who makes a compiler determines the repetition.

(2) A flag indicating whether or not intermediate representations have been changed is prepared. This flag is initially OFF. If intermediate representations are changed during optimization, the flag is turned ON. When the flag is ON, the flag is initialized, and the optimization is repeated.

(3) Optimization is repeated the predetermined number of times, which may be determined by a person who prepares a compiler, or may be optionally set from the outside.

(4) The methods (1) to (3) are combined.

This embodiment employs the method (1). Any one of the repetition methods (1) to (4) may be selected by a user according to an optimization level.

As explained above, the present invention provides an optimizing compiler proper for a given architecture. The optimizing compiler determines a compiler structure, to optimize a source program. The present invention carries out optimization at a proper position according to the characteristics of the source program and architecture, to provide a high-performance object.

The present invention is capable of repeating optimization through compiling phases, to provide a high-performance object.

The present invention carries out the same optimizing and intermediate representation changing processes in respective sections of a compiler, to thereby improve the efficiency of the compiler.

What is claimed is:

1. A compiling apparatus in a data processing apparatus, comprising:

a front end providing intermediate representations described by an intermediate language according to a source program;

optimizing means for optimizing the intermediate representations from the front end in accordance with predetermined optimization functions;

at least one intermediate representation changing means for further transforming the optimized intermediate representations from the optimizing means in accordance with predetermined optimization functions;

code output means for providing code according to the intermediate representations finally obtained by the optimization from the intermediate representation changing means; and

optimizing structure determination means for determining a number of repetitions of an optimization phase achieved by the optimizing means and intermediate representation changing means and selecting the optimization functions for directing an optimizing process carried out in each of the optimization phases, and thereby the intermediate representations being changed and optimized.

2. The compiling apparatus according to claim 1, wherein the optimizing structure determination means determines the number of times of changing the intermediate representations and selects optimization functions to be achieved in each phase according to the intermediate representations dependent on the source program provided by the front end, intermediate representations dependent on an object architecture, and the applied optimization functions.

3. The compiling apparatus according to claim 2, further comprising a switch that connects and disconnects the optimizing structure determination means in

front of the optimizing means and intermediate representation changing means,

the switch being turned ON to determine an optimizing structure in a compiling phase, change intermediate representations according to the source program, and change optimization functions to be applied.

4. The compiling apparatus according to claim 3, wherein, when the switch is turned OFF, the optimizing means and intermediate representation changing means repeat predetermined intermediate representation optimizing and changing processes a predetermined number of times.

5. The compiling apparatus according to claim 4, wherein the optimizing means and intermediate representation changing means have a variable identifying a present phase when repeating the same optimization.

6. The compiling apparatus according to claim 4, further comprising a selection switch to determine whether or not an optional one of the optimization functions must be achieved in an optional one of the optimizing phases, the selection switch being externally controlled to stop an optional optimization function in an optional phase.

7. The compiling apparatus according to claim 3, further comprising a selection switch to determine whether or not an optional one of the optimization functions must be achieved in an optional one of the optimizing phases, the selection switch being externally controlled to stop an optional optimization function in an optional phase.

8. The compiling apparatus according to claim 2, wherein the optimizing means and intermediate representation changing means repeat the same predetermined optimization.

9. The compiling apparatus according to claim 2, wherein a physical memory map structure of main data of the intermediate representations during the intermediate representation changing and optimizing processes is fixed; and

the fixed physical memory map structure are commonly used by the intermediate representation changing and optimizing processes.

10. The compiling apparatus according to claim 2, wherein the optimizing structure determination means selects a plurality of optimizing structures, separately carries out optimization according to each of the selected structures, and selects the best optimizing structure among the optimizing structures according to the optimization results.

11. The compiling apparatus according to claim 1, further comprising a switch that connects and disconnects the optimizing structure determination means in front of the optimizing means and intermediate representation changing means,

the switch being turned ON to determine an optimizing structure in a compiling phase, change intermediate representations according to the source program, and change optimization functions to be applied.

12. The compiling apparatus according to claim 11, wherein, when the switch is turned OFF, the optimizing means and intermediate representation changing means repeat predetermined intermediate representation optimizing and changing processes a predetermined number of times.

13. The compiling apparatus according to claim 12, wherein the optimizing means and intermediate representation changing means have a variable identifying a present phase when repeating the same optimization.

14. The compiling apparatus according to claim 12, further comprising a selection switch to determine whether or not an optional one of the optimization functions must be achieved in an optional one of the optimizing phases, the selection switch being externally controlled to stop an optional optimization function in an optional phase.

15. The compiling apparatus according to claim 11, further comprising a selection switch to determine whether or not an optional one of the optimization functions must be achieved in an optional one of the optimizing phases, the selection switch being externally controlled to stop an optional optimization function in an optional phase.

16. The compiling apparatus according to claim 1, wherein a physical memory map structure of main data of the intermediate representations during the intermediate representation changing and optimizing processes is fixed; and thereby the fixed physical memory map structure are commonly used by the intermediate representation changing and optimizing processes.

17. The compiling apparatus according to claim 1, wherein the optimizing structure determination means selects a plurality of optimizing structures, separately carries out optimization according to each of the selected structures, and selects the best optimizing structure among the optimizing structures according to the optimization results.

18. The compiling apparatus according to claim 1, wherein the optimizing means and intermediate representation changing means repeat the same predetermined optimization.

19. The compiling apparatus according to claim 1, further comprising printing means for printing intermediate representations in each phase and a selection switch for activating the printing means,

the selection switch being externally controlled to print intermediate representations at optional timing in any one of the phases.

20. A compiling apparatus for compiling a source program in a data processing apparatus, comprising:

a front end transforming a source program into intermediate representations;

optimizing means for optimizing the intermediate representations from the front end in accordance with predetermined optimization functions;

at least one intermediate representation changing means for further transferring the optimized intermediate representations from the optimizing means in accordance with predetermined optimization functions, the at least one intermediate representation changing means including final intermediate representation changing means;

code output means for providing code according to the intermediate representations obtained by the optimization from the file in intermediate representation changing means; and

optimizing structure determination means for determining a number of repetitions required for optimization by the optimizing means and intermediate representation changing means, and selecting the optimization functions for directing an optimizing process performed in each of the repetitions, and thereby the intermediate representations being changed and optimized.

21. A compiling method in a data processing apparatus comprising steps of:

a step for providing intermediate representations described by an intermediate language according to a source program;

an optimizing structure determination step for determining the number of repetitions of an optimization phase to execute optimization of the intermediate language and selecting optimization functions to direct an optimizing process carried out in each of the optimization phases;

an optimizing step for optimizing the intermediate representations provided by the source program and the intermediate representations in each of the optimization phases, according to the determination by the optimizing structure determination step;

an intermediate representation changing step for further transforming the optimized intermediate representation in each of the optimization phases, according to the determination by the optimizing structure determination step;

a determination step for determining the number of repetitions of an optimization phase; and

a code output step for providing codes according to the finally obtained intermediate representations.

22. A compiling method according to claim 21, wherein the optimizing structure determination step includes steps of:

a pick up step for selecting at least one optimization function;

a classification step for classifying the selected optimization functions into execution classes; and

a compiler structure determination step for determining the number of times of changing intermediate representations according to the types of intermediate representations and intermediate dependencies, sorting and extracting applicable ones among the optimization functions in each optimization phase, and determining the execution order of the optimization functions according to dependencies of the functions.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,396,631

DATED : March 7, 1995

INVENTOR(S) : Masakazu HAYASHI et al.

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Column 7, Line 56, delete "10".

Column 11, Line 52, delete "10".

Signed and Sealed this

Twenty-fifth Day of April, 1995

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer *Commissioner of Patents and Trademarks*

US005560013A

United States Patent [19]

Scalzi et al.

[11] **Patent Number: 5,560,013**

[45] **Date of Patent: Sep. 24, 1996**

[54] **METHOD OF USING A TARGET PROCESSOR TO EXECUTE PROGRAMS OF A SOURCE ARCHITECTURE THAT USES MULTIPLE ADDRESS SPACES**

[75] Inventors: **Casper A. Scalzi**, Poughkeepsie, N.Y.; **William J. Starke**, Austin, Tex.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **349,772**

[22] Filed: **Dec. 6, 1994**

[51] **Int. Cl.**^{6} **G06F 9/40**

[52] **U.S. Cl.** **395/700**; 395/500; 395/600; 364/280; 364/280.1; 364/228.1; 364/DIG. 1

[58] **Field of Search** 395/700; 396/650; 364/DIG. 1

[56] **References Cited**

U.S. PATENT DOCUMENTS

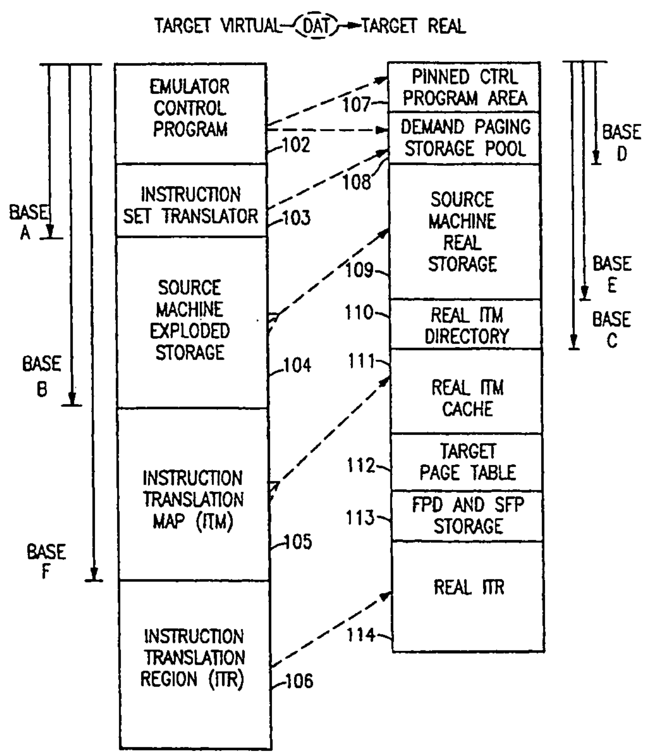
5,313,614	5/1994	Goettelmann et al.	395/500
5,404,478	4/1995	Arai et al.	395/400

Primary Examiner—Kevin A. Kriess
Assistant Examiner—Majid A. Banankhah
Attorney, Agent, or Firm—Bernard M. Goldman

[57] **ABSTRACT**

A method of utilizing large virtual addressing in a target computer to implement an instruction set translator (1ST) for dynamically translating the machine language instructions of an alien source computer into a set of functionally equivalent target computer machine language instructions, providing in the target machine, an execution environment for source machine operating systems, application subsystems, and applications. The target system provides a unique pointer table in target virtual address space that connects each source program instruction in the multiple source virtual address spaces to a target instruction translation which emulates the function of that source instruction in the target system. The target system efficiently stores the translated executable source programs by actually storing only one copy of any source program, regardless of the number of source address spaces in which the source program exists. The target system efficiently manages dynamic changes in the source machine storage, accommodating the nature of a preemptive, multitasking source operating system. The target system preserves the security and data integrity for the source programs on a par with their security and data integrity obtainable when executing in source processors (i.e. having the source architecture as their native architecture). The target computer execution maintains source-architected logical separations between programs and data executing in different source address spaces—without a need for the target system to be aware of the source virtual address spaces.

12 Claims, 13 Drawing Sheets



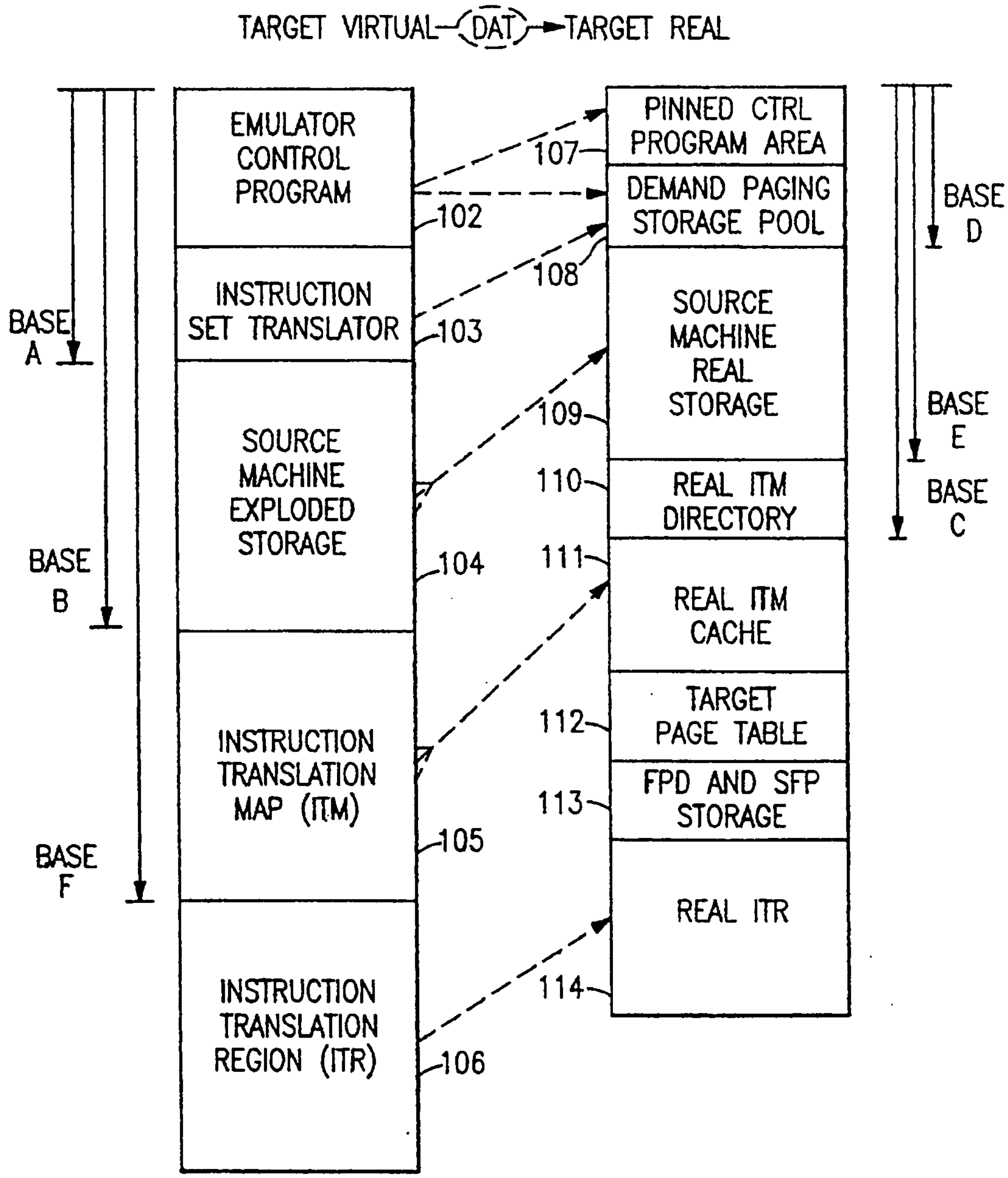

TARGET VIRTUAL
DAT
TARGET REAL
EMULATOR CONTROL PROGRAM
102
INSTRUCTION SET TRANSLATOR
103
SOURCE MACHINE EXPLODED STORAGE
104
INSTRUCTION TRANSLATION MAP (ITM)
105
INSTRUCTION TRANSLATION REGION (ITR)
106
PINNED CTRL PROGRAM AREA
107
DEMAND PAGING STORAGE POOL
108
SOURCE MACHINE REAL STORAGE
109
REAL ITM DIRECTORY
110
REAL ITM CACHE
111
TARGET PAGE TABLE
112
FPD AND SFP STORAGE
113
REAL ITR
114
BASE A
BASE B
BASE F
BASE D
BASE E
BASE C

FIG.1

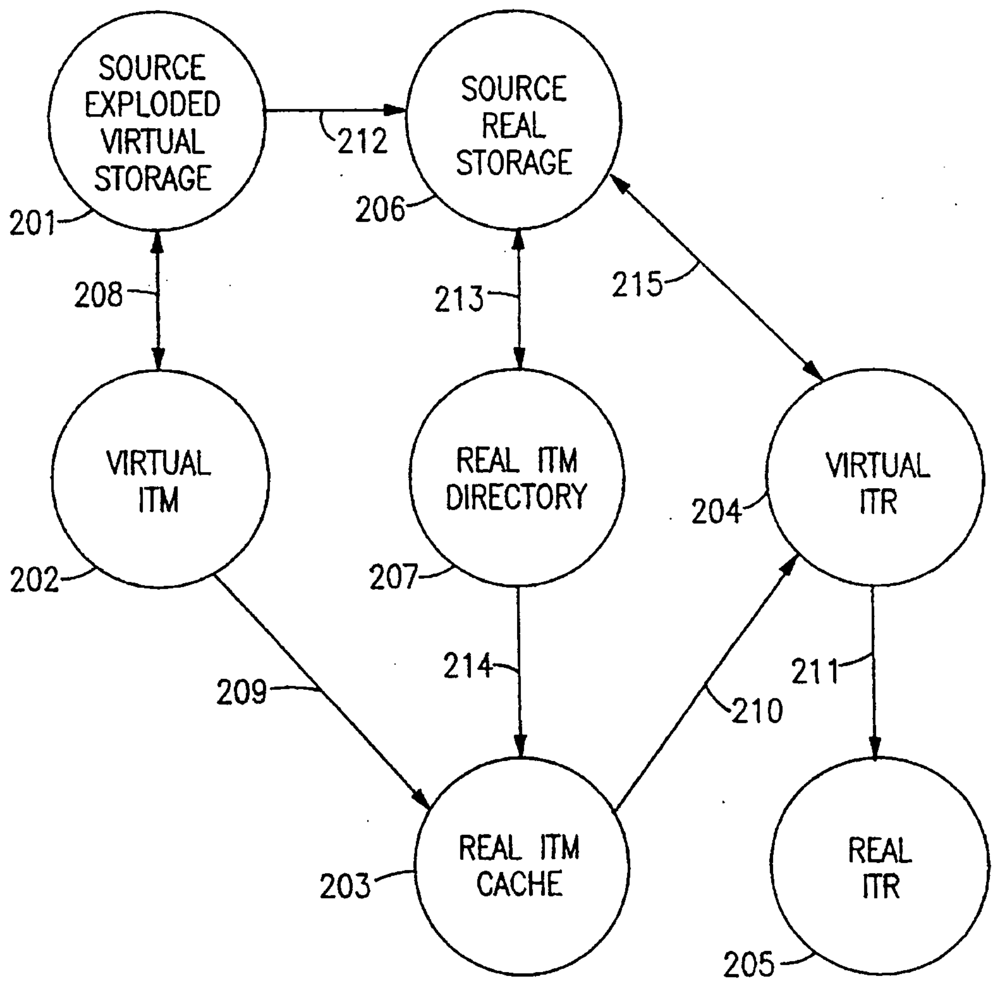

SOURCE EXPLODED VIRTUAL STORAGE
201
212
SOURCE REAL STORAGE
206
208
213
215
VIRTUAL ITM
202
REAL ITM DIRECTORY
207
VIRTUAL ITR
204
209
214
211
210
REAL ITM CACHE
203
REAL ITR
205

FIG.2

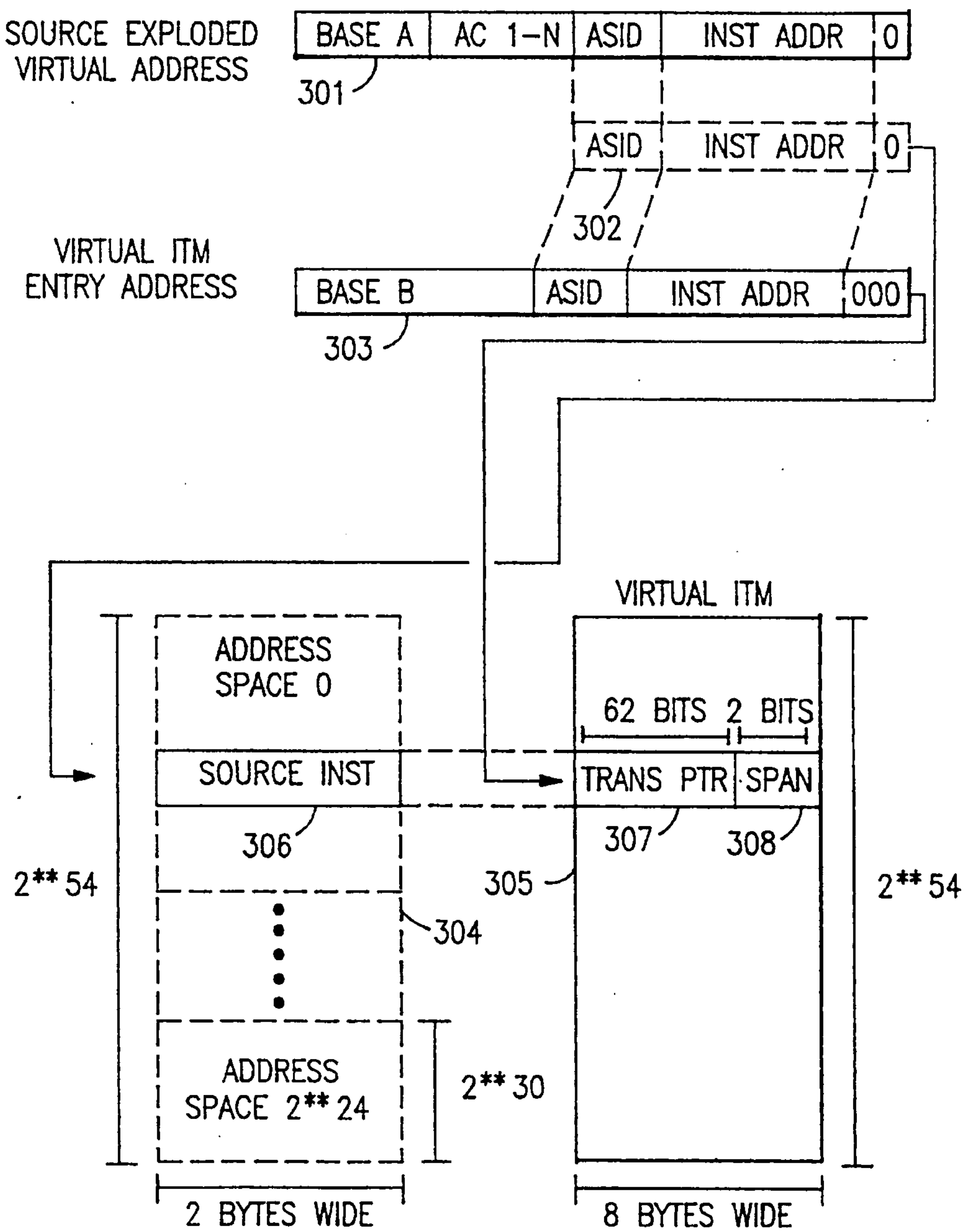

SOURCE EXPLODED VIRTUAL ADDRESS
BASE A
AC 1-N
ASID
INST ADDR
0
301
ASID
INST ADDR
0
302
VIRTUAL ITM ENTRY ADDRESS
BASE B
ASID
INST ADDR
000
303
VIRTUAL ITM
ADDRESS SPACE 0
SOURCE INST
306
304
ADDRESS SPACE 2** 24
2** 54
2** 30
2 BYTES WIDE
62 BITS
2 BITS
TRANS PTR
SPAN
307
308
305
2** 54
8 BYTES WIDE

FIG.3

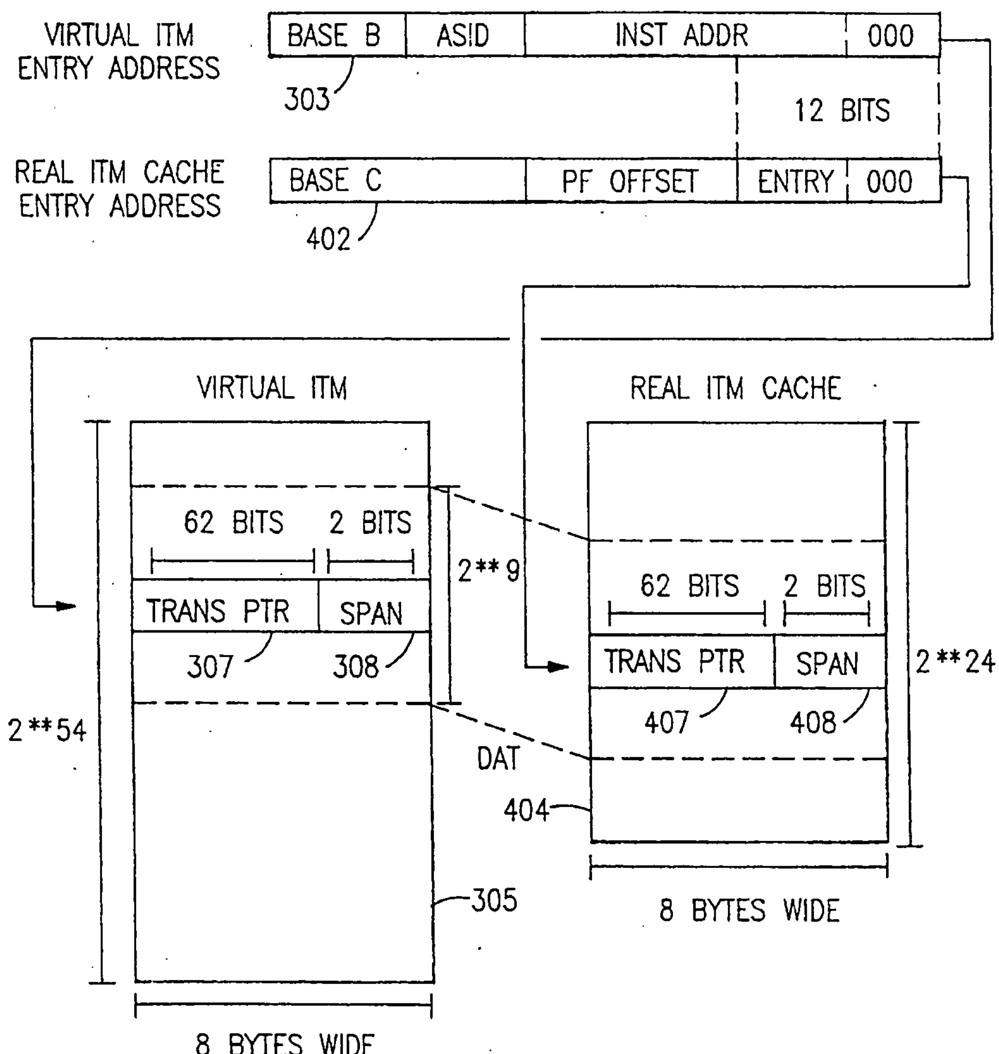

VIRTUAL ITM ENTRY ADDRESS
BASE B
ASID
INST ADDR
000
303
12 BITS
REAL ITM CACHE ENTRY ADDRESS
BASE C
PF OFFSET
ENTRY
000
402
VIRTUAL ITM
62 BITS
2 BITS
TRANS PTR
SPAN
307
308
2**9
2**54
305
8 BYTES WIDE
DAT
REAL ITM CACHE
62 BITS
2 BITS
TRANS PTR
SPAN
407
408
2**24
404
8 BYTES WIDE

FIG.4

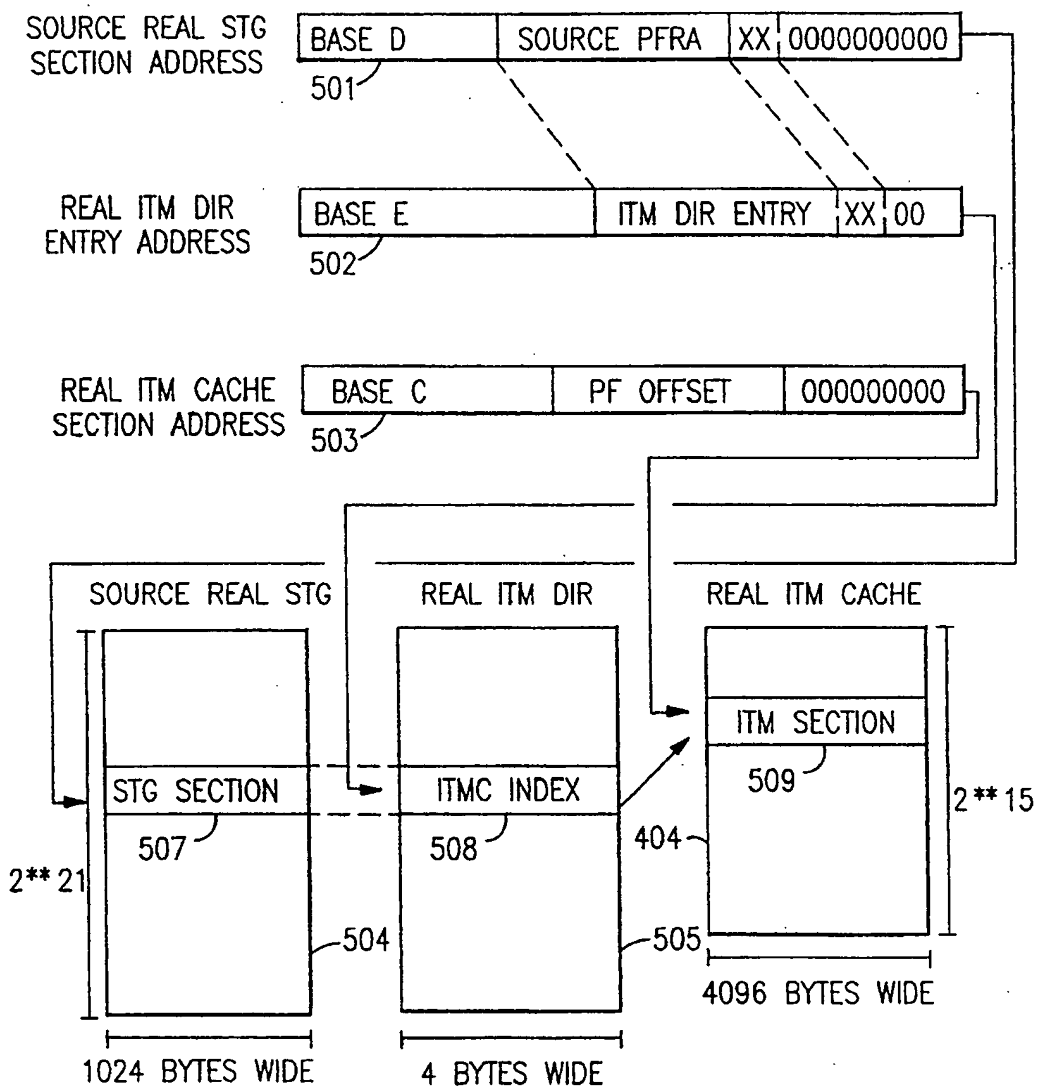

SOURCE REAL STG
SECTION ADDRESS
BASE D
SOURCE PFRA
XX
0000000000
501
REAL ITM DIR
ENTRY ADDRESS
BASE E
ITM DIR ENTRY
XX
00
502
REAL ITM CACHE
SECTION ADDRESS
BASE C
PF OFFSET
000000000
503
SOURCE REAL STG
REAL ITM DIR
REAL ITM CACHE
ITM SECTION
509
2**15
STG SECTION
ITMC INDEX
507
508
404
2**21
504
505
4096 BYTES WIDE
1024 BYTES WIDE
4 BYTES WIDE

FIG.5

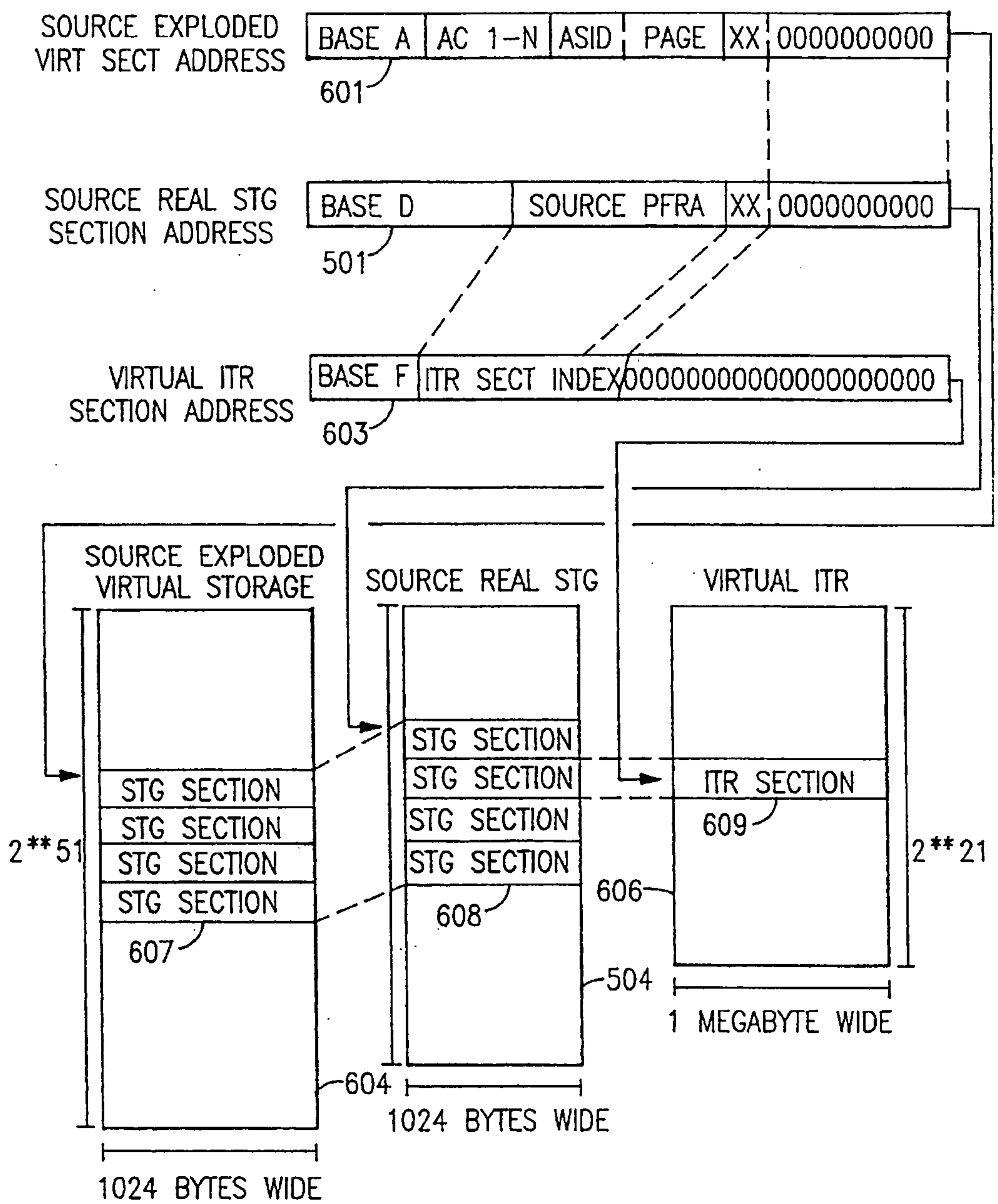

SOURCE EXPLODED VIRT SECT ADDRESS
BASE A
AC 1-N
ASID
PAGE
XX
0000000000
601
SOURCE REAL STG SECTION ADDRESS
BASE D
SOURCE PFRA
XX
0000000000
501
VIRTUAL ITR SECTION ADDRESS
BASE F
ITR SECT INDEX
0000000000000000000
603
SOURCE EXPLODED VIRTUAL STORAGE
SOURCE REAL STG
VIRTUAL ITR
STG SECTION
STG SECTION
STG SECTION
STG SECTION
ITR SECTION
2**51
2**21
607
608
609
606
504
604
1024 BYTES WIDE
1024 BYTES WIDE
1 MEGABYTE WIDE

FIG.6

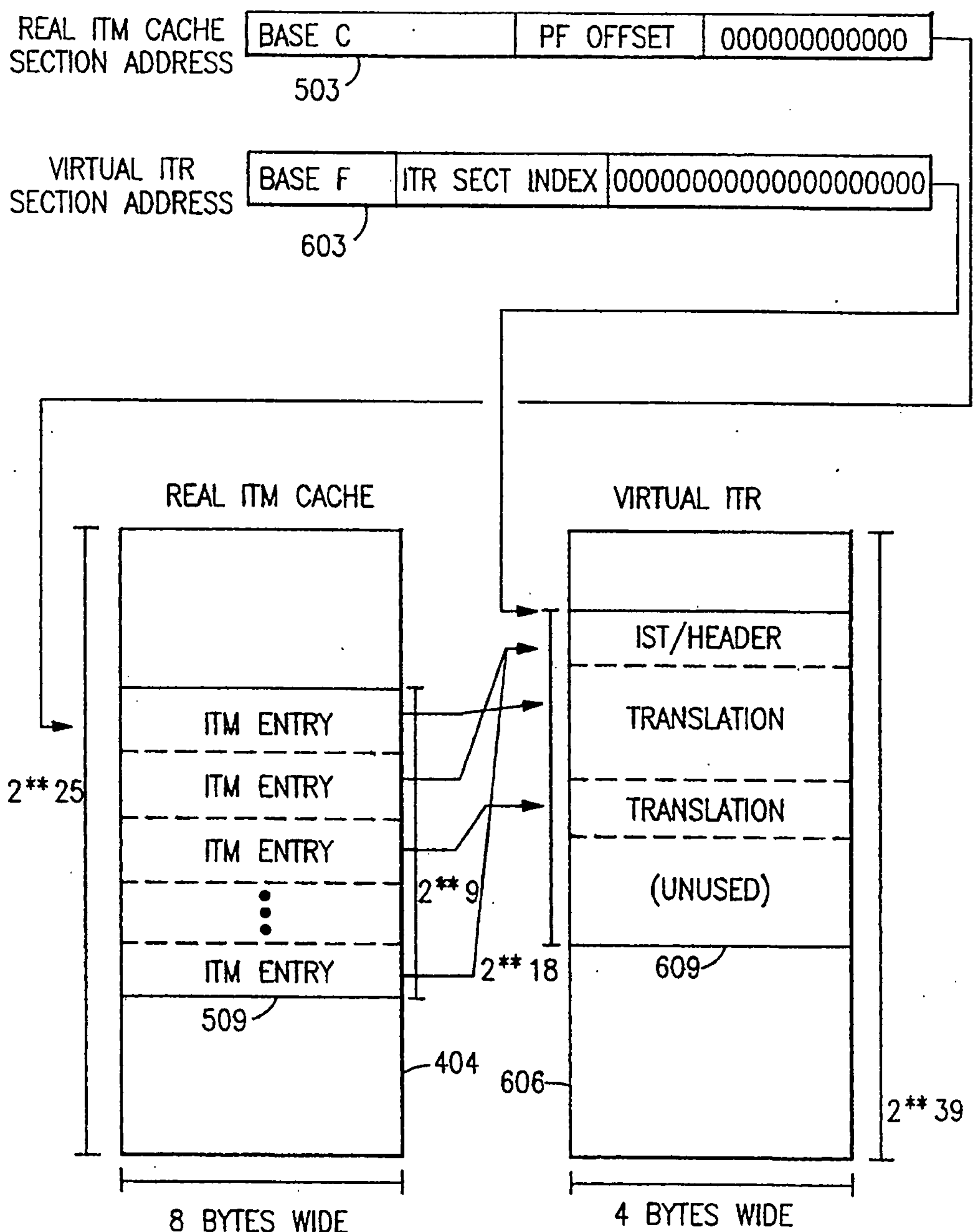

REAL ITM CACHE
SECTION ADDRESS
BASE C
PF OFFSET
000000000000
503
VIRTUAL ITR
SECTION ADDRESS
BASE F
ITR SECT INDEX
00000000000000000000
603
REAL ITM CACHE
VIRTUAL ITR
IST/HEADER
TRANSLATION
TRANSLATION
(UNUSED)
ITM ENTRY
ITM ENTRY
ITM ENTRY
ITM ENTRY
2** 25
2** 9
2** 18
609
509
404
606
2** 39
8 BYTES WIDE
4 BYTES WIDE

FIG.7

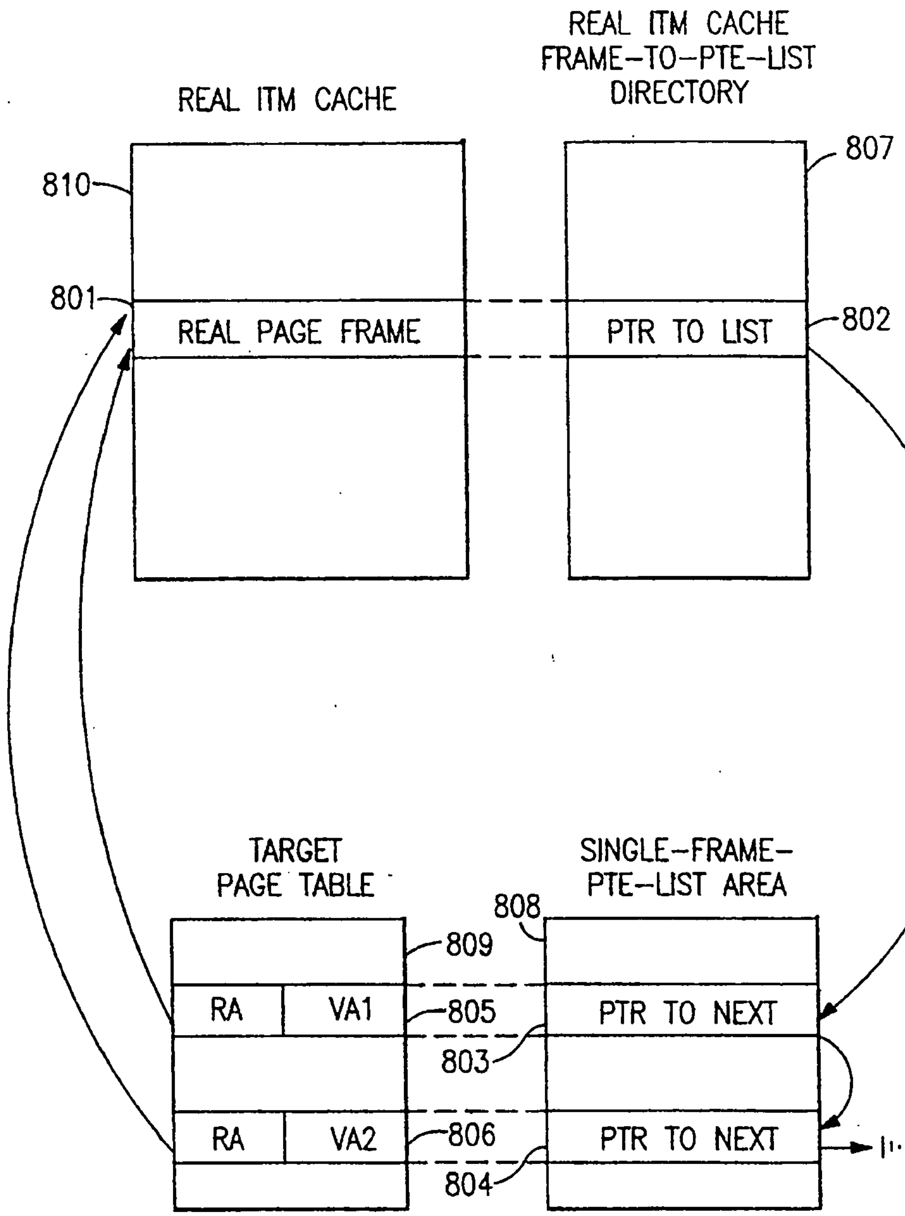
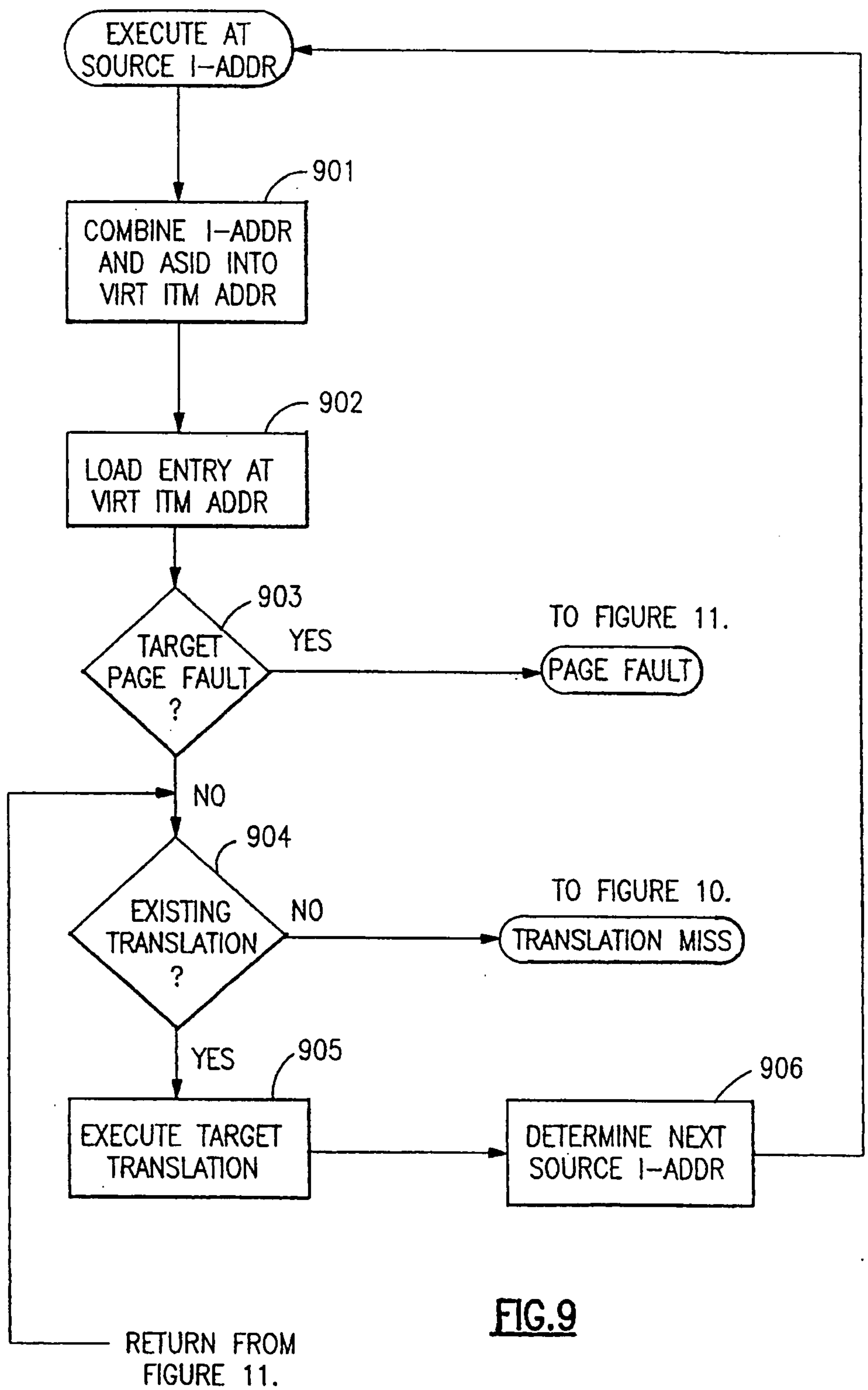

REAL ITM CACHE
REAL ITM CACHE FRAME-TO-PTE-LIST DIRECTORY
807
810
801
REAL PAGE FRAME
PTR TO LIST
802
TARGET PAGE TABLE
SINGLE-FRAME-PTE-LIST AREA
808
809
RA
VA1
805
PTR TO NEXT
803
RA
VA2
806
PTR TO NEXT
804

FIG.8



EXECUTE AT SOURCE I-ADDR
901
COMBINE I-ADDR AND ASID INTO VIRT ITM ADDR
902
LOAD ENTRY AT VIRT ITM ADDR
903
TARGET PAGE FAULT ?
YES
TO FIGURE 11.
PAGE FAULT
NO
904
EXISTING TRANSLATION ?
NO
TO FIGURE 10.
TRANSLATION MISS
YES
905
EXECUTE TARGET TRANSLATION
906
DETERMINE NEXT SOURCE I-ADDR
RETURN FROM FIGURE 11.

FIG.9

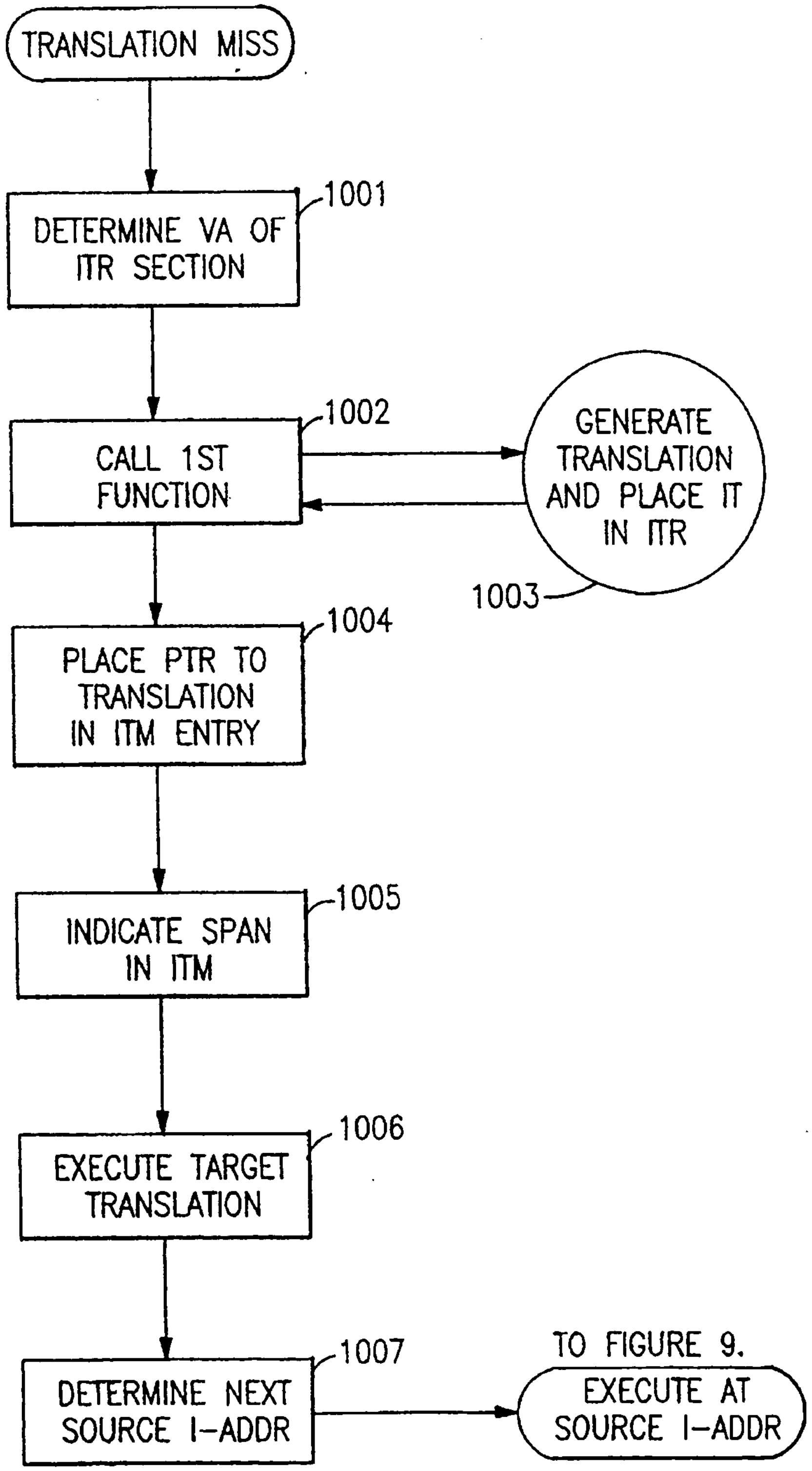

TRANSLATION MISS
DETERMINE VA OF ITR SECTION
1001
CALL 1ST FUNCTION
1002
GENERATE TRANSLATION AND PLACE IT IN ITR
1003
PLACE PTR TO TRANSLATION IN ITM ENTRY
1004
INDICATE SPAN IN ITM
1005
EXECUTE TARGET TRANSLATION
1006
DETERMINE NEXT SOURCE I-ADDR
1007
TO FIGURE 9.
EXECUTE AT SOURCE I-ADDR

FIG.10

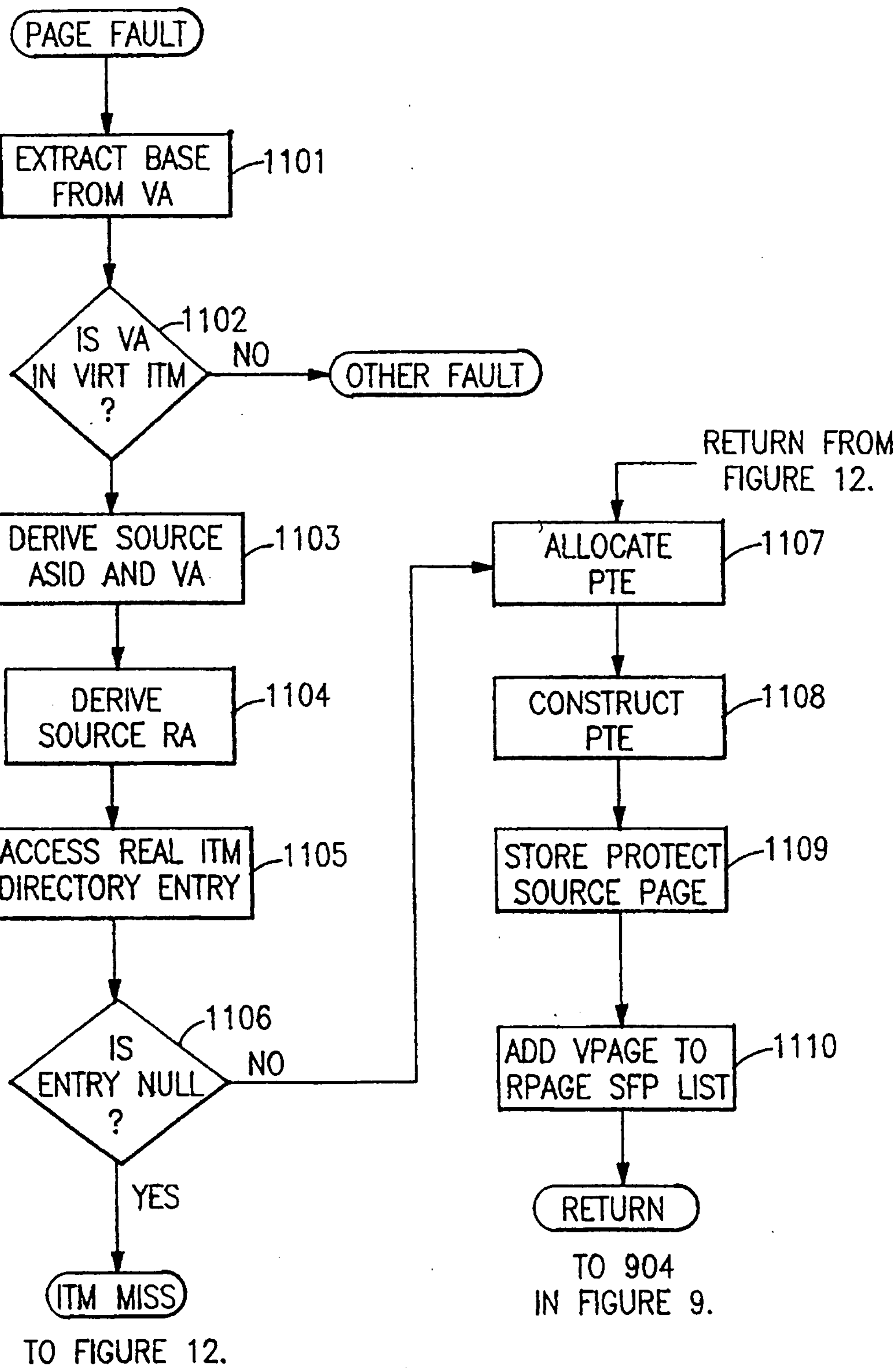

PAGE FAULT
EXTRACT BASE FROM VA
1101
IS VA IN VIRT ITM ?
1102
NO
OTHER FAULT
DERIVE SOURCE ASID AND VA
1103
DERIVE SOURCE RA
1104
ACCESS REAL ITM DIRECTORY ENTRY
1105
IS ENTRY NULL ?
1106
NO
YES
ITM MISS
TO FIGURE 12.
RETURN FROM FIGURE 12.
ALLOCATE PTE
1107
CONSTRUCT PTE
1108
STORE PROTECT SOURCE PAGE
1109
ADD VPAGE TO RPAGE SFP LIST
1110
RETURN
TO 904 IN FIGURE 9.

FIG.11

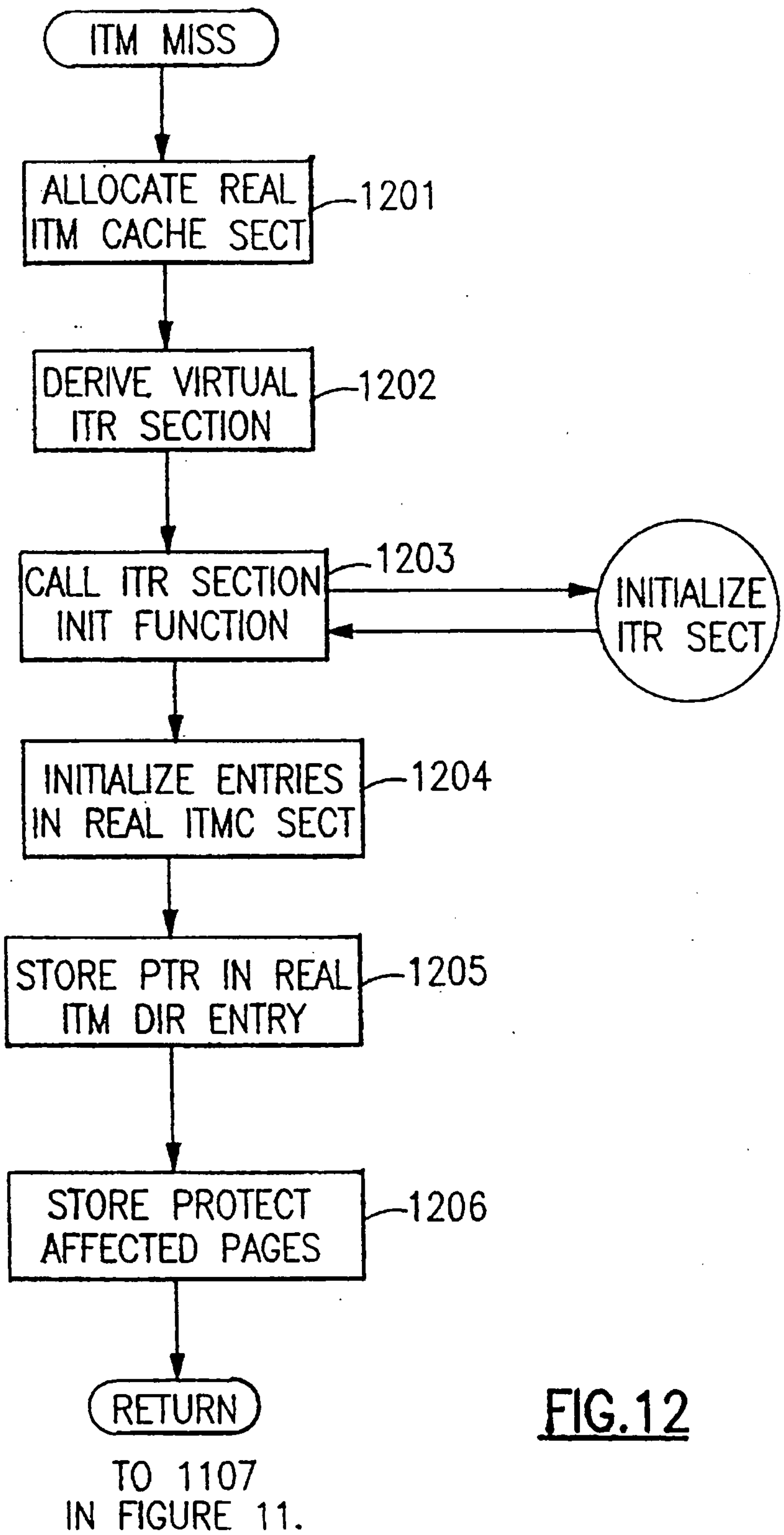

ITM MISS
ALLOCATE REAL ITM CACHE SECT
1201
DERIVE VIRTUAL ITR SECTION
1202
CALL ITR SECTION INIT FUNCTION
1203
INITIALIZE ITR SECT
INITIALIZE ENTRIES IN REAL ITMC SECT
1204
STORE PTR IN REAL ITM DIR ENTRY
1205
STORE PROTECT AFFECTED PAGES
1206
RETURN
TO 1107 IN FIGURE 11.

FIG.12

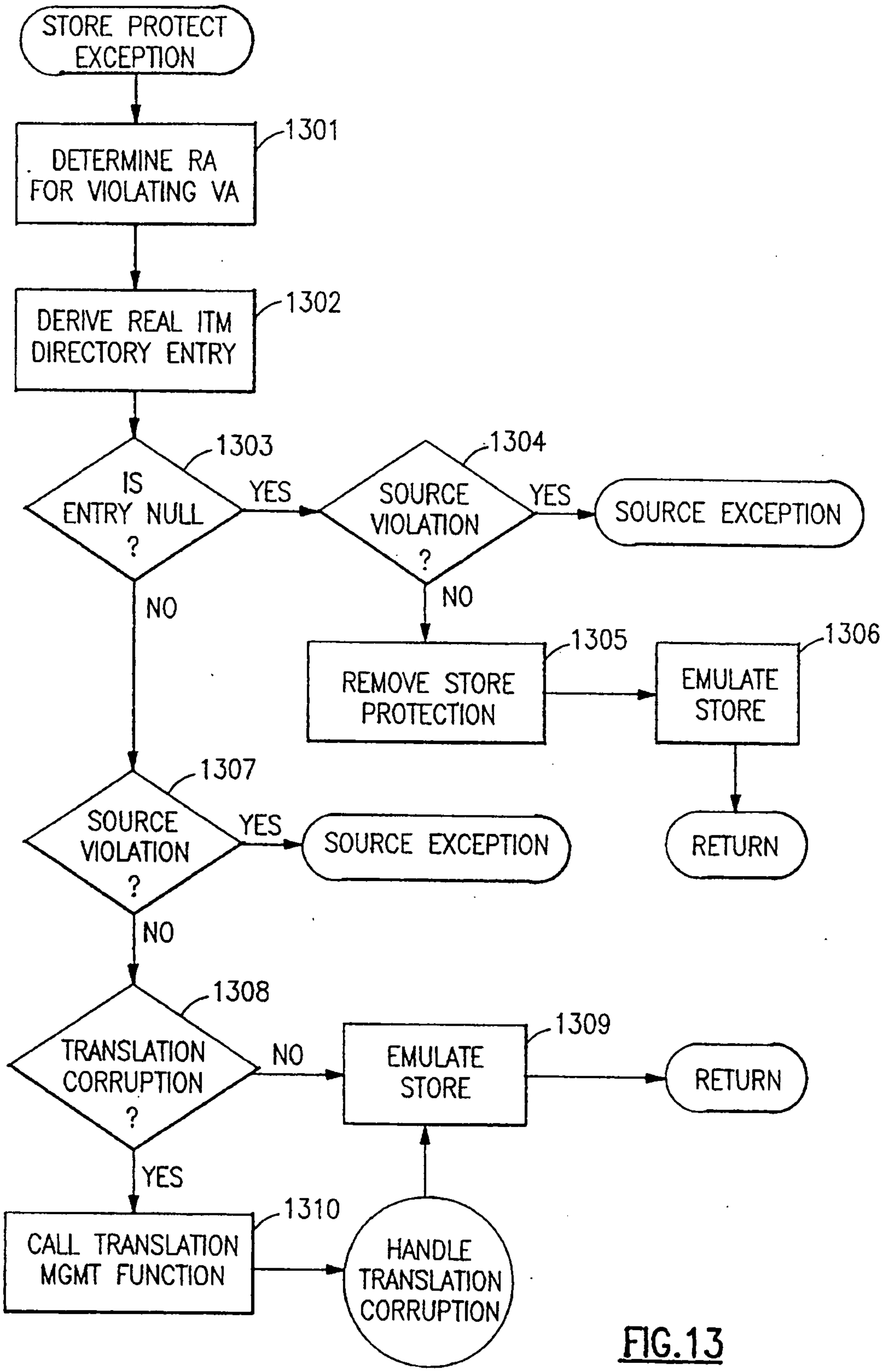

STORE PROTECT EXCEPTION
DETERMINE RA FOR VIOLATING VA
1301
DERIVE REAL ITM DIRECTORY ENTRY
1302
IS ENTRY NULL ?
1303
YES
SOURCE VIOLATION ?
1304
YES
SOURCE EXCEPTION
NO
NO
REMOVE STORE PROTECTION
1305
EMULATE STORE
1306
RETURN
SOURCE VIOLATION ?
1307
YES
SOURCE EXCEPTION
NO
TRANSLATION CORRUPTION ?
1308
NO
EMULATE STORE
1309
RETURN
YES
CALL TRANSLATION MGMT FUNCTION
1310
HANDLE TRANSLATION CORRUPTION

FIG.13

METHOD OF USING A TARGET PROCESSOR TO EXECUTE PROGRAMS OF A SOURCE ARCHITECTURE THAT USES MULTIPLE ADDRESS SPACES

BACKGROUND OF THE INVENTION

The problem of moving programs from one computer architecture to another has given rise to a number of strategies for migration which minimize human effort. Among systems which support the same application interfaces, and exhibit the same security and data integrity characteristics, simple re-compilation of an application may suffice. However, this does not work in cases where high-level language source code is unavailable or where there is no compiler written for the target platform. In these cases, a more general solution is necessary to solve the problem of application migration.

Application migration is not the only problem. Application interfaces are built upon application enablement layers and operating systems, the attributes of which are highly dependent on processor architecture, and thus, may vary greatly from machine to machine. To move an application, it may also be necessary to port the application enablement environment, and possibly the underlying control program. Such an undertaking requires a great deal of human effort, and often fails to preserve the security and data integrity characteristics assumed by the users of the applications, because of operation on the original platform. A general solution to this second problem, which minimizes human effort and preserves desirable architectural attributes, would be quite beneficial in such cases.

A general solution for the first problem, application migration, is provided by an emulation technique known as binary translation, which automatically converts each machine instruction in a program written for a source processor architecture into one or more target processor machine instructions, effectively translating the source machine program, instruction by instruction, into a target machine program.

Often, binary translation is incorporated into a run-time environment, in which a previously unencountered source instruction is dynamically translated into target instructions as needed, and the translation is saved and reused the next time that particular source instruction is to be executed. Of course, data structures must be employed to track which instructions of the source program have been translated, and where the translations are stored. Also, the original source machine program instructions must be maintained, in addition to the target translations of those source instructions. This use of binary translation provides a relatively efficient means for emulating the computational and logical characteristics of a source architecture on a target processor.

Binary translation is far superior to brute force emulation techniques, which interpret source instructions as they are encountered, extracting parameters such as register specifications, and immediate data fields, and calling subroutines based on the opcode. Brute force approaches incur these penalties every time a source instruction is executed, while binary translation techniques incur such penalties only the first time a given instance of a source instruction is executed. Every subsequent execution of that particular instruction instance proceeds with the efficiency afforded by the translated code. In general, these methods have the effect of producing target machine code that has little dependence on source architecture state information, and bears a strong resemblance to code compiled natively for the target architecture.

While binary translation addresses the first problem, providing a general approach for application migration, it does not address the second problem, providing a general approach for migrating the application enablement environment and control program. Binary translation exploits the fact that most processor architectures provide similar computational and logical operations, the aspects of the architecture that are explicitly used by application programs. The layers below the application interface, which provide security and data integrity to the applications, and manage system resources which are shared by multiple independent applications, must use other facilities provided by a processor architecture. These facilities may vary greatly from architecture to architecture, particularily between CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer) architectures, and often the facilities provided by one architecture have no analog in another architecture. The inefficiencies associated with emulating these facilities in software have historically been prohibitive, thereby restricting the use of binary translation to the application domain.

For this reason, other means have generally been used to provide the characteristics of the interface to which emulated applications are written. The software layers underlying this interface are most often rewritten specifically for the target architecture at great expense. Depending on the differences in the underlying processor architectures, many of the security and data integrity characteristics of the environment may be sacrificed in the process. And, unique characteristics of the original operating system can be lost as a result. In other words, no general solution has yet addressed the second problem.

The approach described in application Ser. No. 08/349,771, entitled "Storage Access Authorization Controls In A Computer System Using Dynamic Translation of Large Addresses", assigned to the same assignee as the present invention and filed on the same days as the present invention, incorporated here by reference, does provide a general approach to solving the second problem. By utilizing large virtual addressing in a target processor, it incorporates all the authorization mechanisms of a source processor architecture. The target processor dynamic address translation hardware is used to check the legality of storage accesses, and to map legal storage accesses such that they proceed with the efficiency of the target processor hardware after legality is established. By providing an efficient means to implement the authorization facilities of one architecture on another, it provides the impetus to extend binary translation techniques to cover the entire environment: the application layer, the application enablement environment, and the control program.

It is the goal of this invention to have one computer, the target, provide the necessary program execution environment of another computer, the source, by means of the binary translation of the instructions of the source computer to those of the target computer, such that the control program, application enablement environment, and application programs perform their functions on the target computer in the same manner as they do on the source computer. From the point of view of the computing establishment, the source machine characteristics are observed in the target machine operation, and include the security, integrity, and functional aspects, including those provided by the source operating system, all without the cost of implementing the desired elements of the source as native elements of the target.

Due to their limited scope, existing binary translation techniques are heavily optimized around the issues which exist in the application domain. They do not need to manage the additional complexities which are handled in the software layers beneath the application interface. The approach described in this invention re-focuses the implementation of these techniques, making them feasible for the emulation of the entire system environment.

When the emulated portion of the system environment is extended beyond the application layer, to include the application enablement and control program layers, the ground rules change drastically. The purpose of a multitasking operating system is to maximize the value provided by computing resources, by efficiently distributing them between multiple applications. To do so it must manage the asynchronous events that are part of the total computing environment. As a consequence of such events, the operating system will interrupt the flow of one program and divert computing resources to another. Of course, it must keep track of the current instruction address for the interrupted program, so that the program can resume execution where it left off.

As stated above, most existing binary translation approaches emulate the application domain only, and run on an application interface which runs natively on the target processor. From the perspective of the target operating system, the emulation environment built around each emulated source application is just another target application. Thus, when the multitasking target operating system preempts an emulated source machine program, the return address is actually a target machine address.

However, when the emulated portion of the system environment includes the source operating system, and the interruptions themselves are emulated, the return addresses for preempted programs are source machine addresses. This is because the applications are running under the direct control of the source operating system, which dynamically allocates the system resources among them. Likewise, synchronous and asynchronous exceptions, such as page faults and I/O interrupts, require that a source address be used to determine the point of re-entry into the interrupted program. This is because the source operating system handles these events within the emulated source operating environment. Essentially, even though the application is being executed by means of binary translation, it is executing in a source machine operating environment.

When limited to the application domain, binary translation requires code entry points determined by source addresses only for branches within an application which cannot be determined by static analysis. Existing techniques focus heavily on this, going to great lengths to minimize the number of source entry points. When extended to include the operating system, binary translation requires a code entry point following every possible point of program preemption. This results in a huge increase in the number of possible entry points, specified by a source machine address, in an emulated program. The invention described herein alleviates the severe performance repercussions that such an increase would cause in normal binary translation environments. Also, existing techniques map source instructions in a single application address space to a set of translations for those instructions. When the entire system environment is emulated, there are multiple address spaces, each potentially containing instructions which must be emulated.

The ramifications of dynamic address translation and the possibility of programs shared between multiple address

spaces add complexity and potential for new optimizations, while vastly increasing the scope of the translation management algorithms. For example, some operating system structures map the same system code into all instruction address spaces in order to improve system performance of linkage to, and data access by, that code. If applied to a multi-space emulation environment, traditional techniques would produce multiple translations for the same code, one for each address space that the code is part of. It is advantageous to minimize such duplication. The method described here discovers such multiple mappings of code through use of source DAT, and causes an existing translation of code to be used for all its uses in the source.

Finally, the existing techniques are static in nature, since the code in an application environment is generally not modified during its execution. An entire system environment, on the other hand, is very dynamic in nature. Given the objective of providing the full operational characteristics of the source operating system and those of its native application enablement programs in the emulated environment, full multitasking of multiple applications and application enablement programs must be supported. Application environments are instantiated and discarded. Programs are loaded, and overwritten. The source operating system controls the allocation and deallocation of the assigned source storage, allocated within the target machine storage. Real page frames are allocated in the source storage to back virtual pages, and eventually deallocated. Thus, the algorithms designed to maintain an emulated system environment must support the management of this volatility by creating and destroying target translations as source instructions are created and destroyed in the emulated environment.

SUMMARY OF THE INVENTION

A "source program" is herein defined as a program written for a processor architecture which is alien to the architecture of the target processor. A "source program" contains instructions, operating states and requires executing facilities not found in the target processor architecture. In this specification, the term, source program, is therefore different from, and should not be confused with, a common use of the term "source program" as a program in a high-level language distinguished from the same program in executable object code form. Also, the words machine and processor are often used interchangeably throughout the description of this invention.

As discussed in the background, this invention assumes the methods described in application Ser. No. 08/349,771 for implementing the source processor's architected state and the controls thereof on the target processor. The same requirements on source and target processor attributes are assumed as well, as this invention imposes no additional requirements beyond those described in application Ser. No. 08/349,771.

Application Ser. No. 08/349,771 describes several structures that are established in target machine virtual and target machine real storage that are used in an embodiment of that invention. These include, but are not limited to, target exploded virtual storage, to which all combinations of source effective addresses and storage access states map directly; source real storage, which is represented by a contiguous area of target real storage; the source processor registers, some of which are represented by target processor registers, and others of which are represented within target storage; and the source frame-to-PTE-list directory and

single-frame-PTE-list area, which are used to manage the mapping of multiple virtual pages to a single real page frame. These structures are assumed by this invention.

This invention includes a number of additional structures, which maintain the target machine equivalents of source program instructions, in a manner similar to that of existing binary translation techniques, yet extended to assume the techniques of application Ser. No. 08/349,771, while addressing the issues associated with emulating an entire system operating environment. These structures include an instruction translation map (ITM), an instruction translation region (ITR), and additional structures which are required to effectively maintain the ITM and the ITR. The ITM associates source machine instructions, found in source storage, with the target machine translations of these instructions, which translations are found in the ITR.

FIG. 1 shows a memory map of target machine virtual storage and a memory map of target machine real storage. The important structures used by an emulator which pertain to this invention are shown. The dotted arrows indicate the dynamic address translation (DAT) relationships between structures which exist in virtual storage and the structures in real storage which back them.

This invention relies on a large virtual addressibility in the target machine, as described in the separate application Ser. No. 08/349,771, which is assumed by this invention. The structures which exist in target machine virtual storage are described here.

The emulator control program (**102**) manages the target processor resources as specified in the flowcharts (FIGS. **9** through **13**) of this invention and the flowcharts (FIGS. 26 and 27) of 08/349,771, which this invention assumes, by handling target machine exception conditions and performing various operations on the structures described in this invention and in 08/349,771 as specified herein, as well as managing the target machine page table, and any input/output operations required by a specific embodiment. The emulator control program (**102**) is backed in real storage in part by the pinned control program area (**107**), and in part by the demand paging storage pool (**108**). These methods for backing virtual storage with real storage are well known in the prior art.

The instruction set translator (**103**) is a program which performs several functions related to translating one or more source machine instructions into a set of target machine instructions which perform the equivalent function, and maintaining the translations which it produces. The requirements for the interface to the IST, the types of services provided by the IST, and certain attributes of the target machine instructions produced by the IST are specified in this invention and in 2nd in 08/349,771 (which is assumed by this invention). The portion of the IST which produces target machine translations of source machine instructions is hereafter called the translator function, and the portion of the IST which provided storage management services is hereafter called the translation management function. The IST (**103**) is backed in real storage by page frames from the demand paging storage pool (**108**). This methods for backing virtual storage with real storage is well known in the prior art.

Source machine exploded storage (**104**) is a very large area of target machine virtual storage which area encompasses all of the exploded permutations of all source machine storage locations across all source machine virtual address spaces, the source machine DAT-off address space, and all source machine unresolved address space encodings. It is described in detail in 08/349,771 (which is assumed by this invention), where it is called target exploded virtual storage (since it resides in the target machine's virtual storage). The location of source machine exploded storage (**104**) in target machine virtual storage is indicated by BASE A, which specifies a displacement from target virtual storage address zero. BASE A is referred to later in this invention. Since source machine exploded storage (**104**) is the virtual representation of the contents of the source machine's memory, it is backed in target machine real storage by source machine real storage (**109**), which is a contiguous range of target machine real storage which contains the storage contents of the source machine being emulated. The algorithm which specifies the means and method by which source machine exploded storage pages are backed by source machine real storage page frames is a subject of application Ser. No. 08/349,771, which is assumed in this invention. In the figure, the arrow which indicates the DAT relationship between source machine exploded storage (**104**) and source machine real storage (**109**) is split at the end to indicate that multiple pages in source machine exploded storage (**104**) may be backed by the same page frame in the source machine real storage (**109**) portion of target machine real storage.

This results in complications which are described in application Ser. No. 08/349,771, and later in this invention. Since source machine exploded storage (**104**) is a virtual representation of the entire contents of the source machine's memory, each source machine instruction existing in the source machine's memory will reside at one or more locations in source machine exploded storage (**104**).

The instruction translation map (**105**) occupies a very large range of target machine virtual storage. It is a large table of pointers to instruction translations. There is one such pointer in the ITM for every possible storage location at which an instruction can exist in a source machine address space across all possible source address spaces.

For example, if a source machine supports a 31-bit address, and requires that all instructions begin on a two-byte boundary, then there are 2**(31)/2=2**30=1,073,741,824 possible locations at which a source instruction may reside in a source address space. If the source machine supports a 16-bit address space identifier, then there are 2**(46)=65536 possible address spaces. Thus there are 2**(46)=70,368,744,177,664 possible locations at which an instruction may reside across all source address spaces. It follows that the virtual representation of the ITM in this case would be a table in virtual storage only, containing 70,368,744,177,664 pointers to possible instruction translations. These translations exist in the instruction translation region (**106**) which is backed in real storage by the real ITR (**114**). Due to its immense size and sparcity, the virtual ITM (**105**) is backed by a significantly smaller pool of real storage called the real ITM cache (**111**). The algorithm which specifies the means and method by which virtual ITM pages are backed by real ITM cache page frames is described later in this invention. In the figure, the arrow which indicates the DAT relationship between the virtual ITM (**105**) and real ITM cache (**111**) is split at the end to indicate that multiple pages in the virtual ITM (**105**) may be backed by the same page frame in the real ITM cache (**111**) portion of target machine real storage. This results in complications similar to those described in 08/349,771 for handling source exploded addresses, and later in this invention. The location of the virtual ITM (**105**) in target machine virtual storage is indicated by BASE B, which specifies a displacement from target virtual storage address zero. BASE B is referred to

later in this invention. The ITM is divided into uniformly sized sections which are related to corresponding sections in the instruction translation region (**106**) and source machine real storage (**109**). The relationships between these are described later.

The instruction translation region (**106**) also occupies a very large range of target machine virtual storage. It is divided into uniformly sized sections, such that each instruction translation region (ITR) section maps directly to a corresponding section of source machine real storage (**109**) and to a corresponding section of the ITM (**105**). The details of these relationships are described later. The amount of target machine virtual storage allocated to a virtual ITR section must be sufficiently large to contain all of the target machine translated code associated with all of the source machine instructions in a section of source machine real storage (**109**) in a worst case scenario. There is one such ITR section permanently assigned to each section of source machine real storage (**109**). For example, a particular embodiment may choose to allocate 1,048,576 bytes for each virtual ITR section corresponding to a 1,024 byte source machine real storage section. If the source machine has a total of 2,147,483,648 bytes of real storage, the size of the virtual ITR would be 2,199,023,255,552 bytes. The instruction translations maintained in the virtual ITR (**106**) are located by the pointers contained in the ITM (**105**). Due to its immense size and sparcity, the virtual ITR (**106**) is backed by a significantly smaller pool of real storage called the real ITR (**112**). Real ITR page frames are allocated to virtual ITR pages in a demand paging fashion well known in the prior art. The location of the virtual ITR (**105**) in target machine virtual storage is indicated by BASE F, which specifies a displacement from target virtual storage address zero. BASE F is referred to later in this invention.

The structures which exist in target machine real storage are described here.

The pinned control program area (**107**) is an area of target machine real storage which is permanently assigned ("pinned") to portions of the emulator control program (**102**) and its data structures. Such techniques are well known in the prior art.

The demand paging storage pool (**108**) is an area of target machine real storage from which page frames are allocated as needed to back currently active portions of the emulator control program (**102**), the instruction set translator (**103**), and their associated data structures. Such techniques are well known in the prior art.

Source machine real storage (**109**) is a contiguous range of target machine real storage which contains the real storage contents of the source machine being emulated. Page frames in source machine real storage (**109**) back the virtual pages in source machine exploded storage (**104**). The algorithm which specifies the means and method by which source machine exploded storage pages are backed by source machine real storage page frames is a subject of separate application Ser. No. 08/349,771, which is assumed in this invention. In the figure, the arrow which indicates the DAT relationship between source machine exploded storage (**104**) and source machine real storage (**109**) is split at the end to indicate that multiple pages in source machine exploded storage (**104**) may be backed by the same page frame in the source machine real storage (**109**) portion of target machine real storage. This results in complications which are described in application Ser. No. 08/349,771, and later in this invention. Since source machine real storage (**109**) is a real storage representation of the entire contents of the source machine's memory, each source machine instruction existing in the source machine's memory will reside at one and only one location in source machine real storage (**109**). The location of source machine real storage (**109**) in target machine real storage is indicated by BASE D, which specifies a displacement from target real storage address zero. BASE D is referred to later in this invention. As described earlier, source machine real storage is divided into uniformly sized sections. These are related to corresponding sections in the ITM (**105**) and in the ITR (**106**). The relationships between these are described later. The real ITM directory (**110**) is a table in target real storage which contains one pointer for each section of source machine real storage (**109**). If the pointer contains a NULL value, its corresponding source machine real storage section does not have an associated ITM section in the real ITM cache (**111**). If the pointer is valid, it points to the ITM section in the real ITM cache that is associated with its corresponding source machine real storage section. The real ITM directory exists in target real storage only, i.e., it is only accessed by the emulator when the target machine is in DAT-off mode. The location of the real ITM directory (**110**) in target machine real storage is indicated by BASE E, which specifies a displacement from target real storage address zero. BASE E is referred to later in this invention.

The real ITM cache (**111**) is a pool of real storage which backs the most recently used pages in the virtual ITM (**105**). The algorithm which specifies the means and method by which virtual ITM pages are backed by real ITM cache page frames is described later in this invention. In the figure, the arrow which indicates the DAT relationship between the virtual ITM (**105**) and real ITM cache (**111**) is split at the end to indicate that multiple pages in the virtual ITM (**105**) may be backed by the same page frame in the real ITM cache (**111**) portion of target machine real storage. This results in complications which are described in application Ser. No. 08/349,771, and later in this invention. The location of the real ITM cache (**111**) in target machine real storage is indicated by BASE C, which specifies a displacement from target real storage address zero. BASE C is referred to later in this invention. Since its backs the virtual ITM (**105**), the real ITM cache is also divided into uniformly sized sections. A section in the real ITM cache (**111**) that is associated with a given section of source machine real storage (**109**) may be located by consulting the real ITM directory (**110**). This is described in detail later.

The target machine page table (**112**) exists in target real storage. It contains the page table entries which are used by the target machine DAT mechanism to translate target machine virtual addresses to target machine real addresses. The target machine DAT mechanism is not a subject of this invention, and the invention may embody any target DAT mechanism that does not preclude the association of multiple virtual pages with a single real page frame.

The frame-to-PTE-list directory (FPD) and single-frame-PTE-list (SFP) are structures which are used to manage the mapping of multiple virtual pages to a single real page frame. This is discussed in application Ser. No. 08/349,771, and later in this invention. These structures exist in target real storage (**113**) only, i.e., they are only accessed by the emulator when the target machine is in DAT-off mode.

The real ITR (**114**) is a pool of real storage which backs the most recently used pages in the virtual ITR (**106**). Techniques for backing recently used pages are well known in the prior art.

FIG. **2** shows the relationships among the major structures used by the invention.

Source exploded virtual storage (201) is described in FIG. 1 (104), and in detail in application Ser. No. 08/349,771, which is assumed by this invention. An address in source exploded virtual storage is the input to the process which finds the target machine instruction translation for the source machine instruction which resides at that address. Specifically, the source processor effective instruction address, combined with the address space identifier for the address space from which an instruction is to be fetched indicates a unique source instruction reference from the program's perspective.

The virtual ITM (202) is a table which contains an entry for each possible unique addressable instruction contained in source exploded virtual storage (201). It is assigned a contiguous area of the large target machine virtual address space. Because of this direct, one-to-one mapping, a relationship (208) exists between source exploded virtual storage (201) and the virtual ITM (202), such that given an address into one, the corresponding address in the other may be determined directly. The virtual ITM (202) exists in target virtual storage only, thus, its size requirements, while significant, are accommodated by target machines which meet the large virtual storage requirements enumerated in application Ser. No. 08/349,771. The entries in the virtual ITM (202) are described in the following section which describes the real ITM cache (203).

The real ITM cache (203) is the pool of target real storage which backs the most recently used entries in the virtual ITM (202). Thus, when a target machine page table entry (PTE) exists which maps a page in the virtual ITM (202) to a page frame in the real ITM cache (203), a relationship (209) exists which automatically maps references to the virtual ITM, by target machine dynamic address translation (DAT), to the real ITM cache. When no such PTE exists, the relationship (209) is non-existent. The management of such a situation is described later. Each ITM entry contains two elements. The first element is either a pointer to the set of target instructions which correspond to the source instruction at a given source storage location, or a pointer to the entry point for a function, called the translator, which produces a set of target instructions given a source instruction. The second element contains information related to the span of the source instruction image whose translation is mapped by the entry. The term ITM section is defined to be the unit of granularity by which the real ITM cache (203) backs the virtual ITM (202). For obvious reasons an ITM section must be no smaller than a page of target storage, however, it may be larger, provided that it is an exact multiple of the target page size. The term source storage section is defined to be a unit of source storage that directly corresponds to an ITM section. For example, if source instructions are addressable on two-byte boundaries, and ITM entries are eight bytes in size, and the size of an ITM section is set at 4096 bytes, the size of a source storage section would be 1024 bytes. The term ITR section is defined to be a unit of ITR virtual storage that directly corresponds to an ITM section and a source storage section. The ITR section is described fully below.

The virtual ITR (204) is a large area of target virtual storage which contains the sets of target instructions or translator function entry points referenced by the first element of each entry in the real ITM cache (203). The virtual ITR (204) is divided into ITR sections that directly correspond to ITM sections and source storage sections. A direct, one-to-one relationship (215) exists between the virtual ITR (204) and source real storage (206), such that each source storage section in source real storage (206), has a corresponding ITR section in the virtual ITR (204). Therefore, ITR sections may be arbitrarily large, as long as the virtual ITR (204) contains one ITR section for each source storage section in source real storage (206). An ITR section should be large enough to hold the translations for all possible source instructions in the source storage section associated with it. The internal organization and storage management of the virtual ITR is described later. When an entry in the real ITM cache (203) holds a pointer to a set of target instructions which correspond to a source instruction, which target instructions will be found in the ITR section associated with the ITM section, a relationship (210) exists through which a pointer in the real ITM cache is used to access an instruction translation. When an entry holds a pointer to the translator function for the ITR section, which translator function entry point will be found in the ITR section associated with the ITM section, the relationship (210) is non-existent. The management of such a situation is described later.

The real ITR (205) is the pool of target real storage which backs the most recently used pages in the virtual ITR (204). Thus, when a target machine page table entry (PTE) exists which maps a page in the virtual ITR (204) to a page frame in the real ITR (205), a relationship (211) exists which automatically maps references to the virtual ITR, by target machine dynamic address translation (DAT), to the real ITR. When no such PTE exists, the relationship (209) is non-existent. The management of such a situation is described later.

Source real storage (206) is the contiguous block of target real storage which directly represents source machine real storage. It is the ultimate repository of source programs, source data, and source control structures such as page tables. Since it either backs all source virtual storage pages or contains the structures used by the algorithms which back all source virtual storage pages on demand, there is a relationship (212), by which references to source exploded virtual storage (201) are mapped to source real storage (206) through the use of the source DAT mechanisms. As described above, there is a direct, one-to-one relationship (215) between each source storage section in source real storage, and its corresponding ITR section in the virtual ITR (204). That is, the translations for the instructions found in a source storage section in source real storage (206) are maintained in the corresponding ITR section found in the virtual ITR (204), backed by pages from the real ITR (205).

The real ITM directory (207) is a table, which exists in target real storage, that contains an entry for each source storage section in source real storage (206). Because of this direct, one-to-one mapping, a relationship (213) exists between source real storage (206) and the real ITM directory (207). Each entry in the real ITM directory (207) contains either a pointer to the ITM section in the real ITM cache (203) which contains the set of entries which map the source instruction images contained in the source storage section found in source real storage (206) which corresponds to the entry in the real ITM directory (207), or a NULL pointer if there is no ITM section in the real ITM cache (203) corresponding to the source storage section in source real storage (206). In the former case, a relationship (214) exists through which a pointer in the real ITM directory (207) is used to access an ITM section in the real ITM cache (203). In the NULL case, the relationship (214) is non-existent. The management of such a situation is described later.

For simplicity the real ITM cache (203) and the real ITR (205) have been described as independent pools of real storage, dedicated to backing their respective virtual struc-

tures. Depending on the embodiment, however, it is likely that the real ITR (205), and possibly the real ITM cache (203) may be drawn from a common, demand-paging pool of real storage in the target machine. However, source real storage (206) and the real ITM directory (207) must be independent contiguous areas of target real storage.

The goal of efficiently locating a set of target machine instructions associated with a source machine effective instruction address is attained as described below. It is assumed in this process that any source page fault caused by an access to the source instruction being executed has been resolved, i.e., that relationship (212) exists. This may be tested by employing the techniques described in application Ser. No. 08/349,771 to perform an emulated source instruction fetch. As stated above, the source effective instruction address combined with the source address space identifier for the source address space in which the source instruction exists (both of which may be extracted from the source exploded virtual address) yield an index into the virtual ITM (202). When the DAT relationship (209) exists (it does in steady-state operation), an address into the real ITM cache (203) is produced by the target machine DAT hardware. The case in which the DAT relationship (209) does not exist is called an ITM fault, and is described later. In the case where the DAT relationship (209) does exist, the referenced real ITM cache entry contains a pointer. When this pointer references the set of target instructions that make up the translation of the source instruction (the steady-state case), i.e., the relationship (210) exists, the address into the virtual ITR (204) is known. The case in which the relationship (210) does not exist is called a translation miss, and is described later. If the relationship (210) does exist, the address into the virtual ITR is converted, by target machine DAT, into an address into the real ITR. If the DAT relationship (211) exists, the process is complete, and the set of target instructions may be executed. If the relationship (211) does not exist, an ITR fault occurs, and the target machine must allocate a frame from the real ITR to back the virtual page.

In the event of a translation miss, i.e., relationship (210) does not exist, the pointer in the ITM points to the translator function entry point in the ITR section instead of a translation. The translator function (or IST) has the responsibility of producing a set of target instructions to perform the function of the source instruction, managing the placement of these instructions into the virtual ITR section associated with the source real storage section from which the source instruction image was obtained, replacing the reference to itself in the ITM, with a pointer to the newly created target instructions, and indicating the span of the translation in the ITM. This action establishes relationship (210), and completes the process of finding the desired set of target instructions.

In the event of an ITM fault, i.e., relationship (209) does not exist, the target machine observes a page fault, and the appropriate target machine exception handler is invoked. The exception handler determines, from the fact that the faulting virtual address referenced the area of target virtual storage allocated to the virtual ITM, that an ITM fault has occurred. The faulting address into the virtual ITM (202) is converted, by direct relationship (208), to an address into source exploded storage (201), which is converted, by source DAT relationship (212), to an address into source real storage (206). Note that when an ITM fault occurs, this DAT relationship (212) is guaranteed to exist, as stated above. The address into source real storage (206) is converted, by direct relationship (213), to an address into the real ITM directory (207). If the real ITM directory entry referenced by this address contains a NULL pointer, the pointer relationship (214) does not exist. This situation is called an ITM miss, and is described later. If the relationship (214) does exist, the pointer in the real ITM directory entry points to the ITM section in the real ITM cache (203) which is to back the ITM section in the virtual ITM (202) which caused the ITM fault. One or more target PTE's (depending on the size of an ITM section) are allocated to establish the DAT relationship (209) between the faulting ITM section in the virtual ITM (202) and the ITM section in the real ITM cache (203). As a means to detect modification of the instruction stream, a control mechanism must be implemented using either the methods described in application Ser. No. 08/349,771 or a native store protection mechanism inherent in the target processor architecture, to protect the emulated source storage pages related to the virtual ITM section from stores. This will cause any stores to source storage that is associated with an ITM section that exists in the real ITM cache (203), to result in a target exception, so that appropriate action may be taken to update the ITM and ITR when source instruction images are modified. Such a control mechanism is described later. Once relationship (209) is established and the source storage protections are in place, the original interrupted process can resume.

In the event of an ITM miss, i.e., relationship (214) does not exist, there is no ITM section in the real ITM cache (203) to back the ITM section in the virtual ITM (202), so one must be allocated. Normally such an allocation requires that an old ITM section in the real ITM cache (203) be destroyed to make room for the new ITM section. The algorithm used to determine which ITM section to destroy is not a subject of this invention, although it is recommended that this algorithm employ some kind of least-recently-used (LRU) scheme, such as testing the reference bits associated with pages of storage. When destroying the ITM section in the real ITM cache (203), the real ITM directory (207) entry associated with the old ITM section must be set to NULL. This is done by looking up any of the entries in the ITM section to determine the ITR section, and from that, to determine the source real storage section, and from its address, to determine the real ITM directory entry. Also, all virtual mappings to the old ITM section must be destroyed as well. Since it is possible that multiple mappings may exist, it is necessary to keep track of all virtual ITM (202) pages associated with a given real page frame in the real ITM cache (203). This may be accomplished using the techniques for managing multiple-virtual-to-single-real page mappings described in application Ser. No. 08/349,771. A particular embodiment may determine that, at this time, it is advantageous to remove the store protection from the source storage associated with the ITM section. This, however, is not required, as a particular embodiment may decide to wait until an exception occurs and, at that time, upon discovering that the store protection exists for no apparent reason, the embodiment may remove it. It is also possible, though not required, to free real page frames that back virtual pages in the virtual ITR (204) that are referenced by the ITM entries in the ITM section that is being destroyed, since the information they contain is no longer relevant. Once the old ITM section is destroyed, a new one may be created. This is accomplished by using the address into source real storage (206) which was determined during the resolution of the initial ITM fault, and converting it by direct relationship (215), to the address into the virtual ITR (204). This address is rounded down to the beginning of an ITR section. An ITR header structure is placed at this location. An ITR header structure is used to provide a unique

entry point to the translator function, which is associated with a particular ITR section, and to provide the state information used by the translator function to manage the translations maintained in that ITR section. The organization of the ITR header structure is determined by the translation management function. Note that storing the ITR header structure may cause a target page fault, which must be managed in normal fashion by demand paging from the real ITR (**205**). All entries in the new ITM section are initialized to point to the translator function entry in the ITR header structure. The real storage address of the new ITM section (in the real ITM cache) is placed in the real ITM directory (**207**) entry which was determined during the resolution of the initial ITM fault, thus establishing the pointer relationship (**214**). It is necessary to apply the storage protection mechanisms referenced earlier to emulated source storage page(s) associated with the new ITM section. Once relationship (**214**) is established and the source storage protections are in place, the ITM fault process can resume.

The goal of maintaining a single set of translations for programs that are mapped into multiple source effective address spaces is attained by taking advantage of the fact that such duplication exists only in source exploded virtual storage (**201**), and not in source real storage (**206**), which is the ultimate repository for source machine instructions. By associating the target translations of source instructions, which translations are found in the virtual ITR (**204**), with the mapping of those source instructions in source real storage (**206**), unnecessary duplication can be eliminated. Thus, relationship (**215**) establishes this association of instruction images in source real storage (**206**) to their target machine equivalents in the virtual ITR (**204**). At odds with this, is the requirement that a translation must be located given the address of its corresponding source instruction in source exploded virtual storage (**201**). There may be many such source exploded virtual locations which are backed by the same source real location, and are thus associated with the same translation. Therefore, the ITM, which associates source instruction images with their translations in the ITR, must index the translations by components of source exploded virtual addresses, while the translations themselves are associated with source real addresses. Another important consideration is minimizing the target real storage requirements of the ITM, by eliminating duplication there as well.

An optimal solution is attained by utilizing the free association provided by target DAT to break the ITM into two representations: a virtual representation, the virtual ITM (**202**), which has a direct relationship (**208**) to source exploded virtual storage (**201**); and a real representation, the real ITM cache (**203**), in which each real ITM cache (**203**) section has a one-to-one relationship (**210**) with a virtual ITR (**204**) section, which, in turn, has a one-to-one relationship (**215**) with a source real storage (**206**) section. Thus, for each section of source real storage, there is at most one set of instruction translations in the ITR, and at most one ITM section in the real ITM cache (**203**). In other words, regardless of the number of source address spaces that contain a particular set of instructions, only one set of translations exists in the target real storage, and all virtual ITM sections for the particular set of instructions are backed by the same real ITM cache section. This minimizes target real storage consumption. Target DAT provides the relationship (**209**) which ties together the virtual ITM (**202**) and the real ITM cache (**203**). The techniques described in application Ser. No. 08/349,771 are employed to manage the multiple-virtual-to-single-real mappings and the real ITM directory (**207**) is used as described earlier to create the mappings.

The goal of managing the volatility of an emulated source machine operating environment is attained by incorporating the ability to detect changes in that environment, and the means to manage system resources so as to react to those changes in an efficient manner. This is closely related to the goal of an operating system, and, in fact, the emulated source machine operating system will address these issues insofar as they pertain to those portions of the target machine that represent source machine facilities. The ITM and ITR are not source machine facilities, and the source operating system has no knowledge of them. They are, however, affected by the actions of the source machine, and therefore it is necessary to detect such source machine actions and reflect their consequences in the ITM and ITR. At the same time, the target machine must manage the resources used by the ITM and ITR as efficiently as possible.

The translations maintained in the ITR are dependent on the instruction images in source real storage (**206**). Therefore, if a translation exists in the ITR for a given source instruction image, and that source instruction image is modified by a source machine operation, such modification must be detected, and the ITM and ITR must be updated accordingly, to reflect the effects of the modification.

To this end, a store protection mechanism, provided either by the techniques described in application Ser. No. 08/349,771, or by the target machine hardware, is employed, which causes a target machine exception whenever a store to a protected page occurs. As in application Ser. No. 08/349,771, storage references can be qualified in target exploded virtual addresses by whether they are fetches or stores. Fetches can be allowed to operate at full processor speed. Stores to source pages containing translated instructions can be trapped by a page fault on the exploded address, and checked to determine whether or not they will modify a translated instruction. On the other hand, a target processor may contain a facility to protect a page from a store while allowing fetches to proceed without interference. Such a mechanism may protect a real storage page frame, as does the storage key of IBM S/390, or it may protect a virtual page by providing the specification of store protection in a page table entry, as in the PowerPC architecture. This invention requires that stores into already translated source instructions be detected, but is not dependent on the specific mechanism used to do so. The methods described provide correct functional operation of the total source operating environment regardless of such changes in instructions. This is achieved by the dynamic binary translation of the instructions not yet translated, and the detection of stores into already translated instructions and the invalidation of affected translations. The protection is applied to all pages which represent source machine pages for which an ITM section resides in the real ITM cache (**203**). This is accomplished by identifying all such pages at the time when the ITM section is created (refer to ITM misses), and, if necessary, by protecting virtual pages or their backing page frames as they are added to the set of such pages (refer to ITM faults).

When such an exception occurs, the target machine exception handler must first determine if the exception was caused by storing into a page containing instructions for which an ITM section and ITR region currently exist. This can be accomplished quickly, by determining the real ITM directory (**207**) entry associated with the source real storage backing the store which caused the exception. If the entry contains a NULL pointer, there is no ITM section associated with the storage. This points to one of two possibilities: either a source authorization mechanism caused the excep-

tion, or there was previously an ITM section associated with the source page, and when that ITM section was destroyed, the store protection was not removed. If the real ITM directory (**207**) entry contains a valid pointer, there is an ITM section associated with the storage. In this case there are also two possibilities: there may or may not be a source storage authorization violation in addition to the ITM related exception. Thus, regardless of whether the ITM section exists, the methods of application Ser. No. 08/349,771 must be used to determine whether or not a source storage authorization violation has occurred. If so, the appropriate source exception must be emulated in the source operating environment. If not, and no ITM section exists, the store protection can be removed, and the violating store re-executed. If not, and an ITM section does exist, the individual ITM entries affected by the store must be checked. This case, in which pointer and span information in these entries indicates that a translation has been affected by an overwrite of all or part of a source instruction, is called a translation corruption, and is described later. In the case in which the pointers in affected entries point to the translator function, no translations are affected, and the target machine can simply emulate the effects of the store, and return control to the instruction following the store.

In the event of a translation corruption, the ITM and ITR must be updated to reflect the change in source machine storage. The span field of the ITM can be used to calculate the scope of a store operation into a program. If any part of an already translated source instruction is altered during source machine execution, either its ITM entry must be reset to return to the translator if the translation is subsequently executed, or the real ITM section can be deallocated. Specific actions in consequence of such an invalidation of an existing translation are the responsibility of the translation management function. Regardless of the action of the translation manager, control must finally be returned to the original exception handler, to emulate the modification of source machine storage.

As a result of this kind of storage volatility, it may not be wise to expend a great deal of effort optimizing the translated code. Due to the shorter life span of translations in a fully functional source operating system, the fixed overhead associated with optimizing a translation contributes to a higher per-execution overhead. One of the effects of less optimization is an increase in the number of source machine code entry points. Without the efficient lookup mechanisms described earlier, including beneficial use of the large virtual address space of the target processor, such an increase would produce unfavorable performance implications. Thus, the benefit realized by these mechanisms is increased by the effects of the storage volatility in the full source operating environment emulation.

It is an object of this invention to have one computer, called the target, provide the complete program execution environment of another computer, called the source, by means of binary translation of the instructions of the source computer to those of the target computer such that the control program, the application enablement programs, and the application programs of the source execute on the target as they do on the source without change to any of the programs.

It is another object of this invention to provide the program execution environment of a source computer on a different target computer, using binary translation of source instructions to target instructions, and utilizing the methods of application Ser. No. 08/349,771 to provide the storage access authority states of the source in the target operation of source programs.

It is another object of this invention to use the large virtual address space of the target machine to provide a direct addressing relationship, requiring a minimum number of steps for access, between source virtual addresses and an instruction translation map indicating target machine instruction translations of the source instructions at those addresses.

It is another object of this invention to use the large virtual address space of the target machine to provide a direct addressing relationship, requiring a minimum number of steps for access, between each section of source real storage within the target and the section of target virtual storage which contains the target instruction translations instructions in the section of source real storage.

It is another object of this invention to use the large virtual address space of the target machine to provide a direct-address map of source machine instructions to target machine instruction translations, in addition to its use for providing the storage addressing required by the source programming execution environment, which includes those source instructions.

It is another object of this invention to back in target real storage the direct virtually-addressed map that associates source instructions with the corresponding target instruction translations in such a way that those portions of the map which associate source instructions that are common to multiple source address spaces with corresponding target translations are backed by the same target real storage.

It is another object of this invention to maintain a single copy in target real storage of the target machine instruction translations of source instructions that are common to multiple source virtual address spaces.

It is another object of this invention to use target machine real storage to back only the sections of the direct map of source machine instructions to target machine translations for those instruction translations currently being maintained, indicating these sections by use of a directory which associates source real storage sections with the target real storage map sections for target translations of source instructions in those source real storage sections.

It is another object of this invention to use target real storage to back only the sections of the target translations of source instructions for which a map section exists in target real storage, which associates source instructions with valid target translations.

It is another object of this invention to use the method of application Ser. No. 08/349,771 to relate all map sections in target virtual storage backed by a single map section in target real storage, because they are for the same source machine instructions in multiple source address spaces, such that given any specific map section in target real storage, all map sections in target virtual storage that it backs can be determined.

It is another object of this invention to impose store protection on target storage containing source machine instructions whose target translations are currently being maintained, by using either the methods described in application Ser. No. 08/349,771 or the native facilities of the target machine, such that modifications made by a source machine program to such instructions are detected.

It is another object of this invention to record the source machine storage span represented by a target machine translation of a source instruction, such that, on source machine storage operations modifying any part of already translated source instructions, it can be determined which translation or translations must be discarded, if any.

It is another object of this invention to use the large virtual address space of a target machine to provide a unique area of target machine virtual storage for each unique section of source machine real storage, such that this unique area of target machine virtual storage is of sufficient size to hold the target machine translations for every source machine instruction found in the source machine real storage section, and that this unique area of target machine virtual storage may be backed in target machine real storage by demand paging to manage the amount of real storage required to store the translations.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows the virtual storage and real storage maps of the the target machine processor.

FIG. 2 shows the addressing relationships among the major structures used in this invention.

FIG. 3 illustrates the addressing relationship between a source instruction address in source exploded virtual form and the virtual address of the ITM entry that indicates its translation state.

FIG. 4 illustrates the relationship between an ITM virtual section and a backing real ITM cache section.

FIG. 5 illustrates how the address of an existing backing real ITM section can be found when a page fault occurs on a virtual ITM reference.

FIG. 6 shows the relationship between source real storage sections and target virtual ITR sections.

FIG. 7 shows the relationship real ITM entries and the ITR sections that contain instruction translations associated with the ITM entries.

FIG. 8 shows the structure that associates target PTEs resolving different virtual ITM sections to the same real ITM section.

FIG. 9 shows the target processor overall logic for executing a source instruction.

FIG. 10 illustrates the target processor logic for handing an instruction translation miss occurs.

FIG. 11 shows the processing performed when a page fault occurs on a reference to an ITM entry.

FIG. 12 shows the processing when a real ITM section must be assigned to back a virtual ITM section.

FIG. 13 shows the process for a source program store operation into a source real storage section from which some instructions have already been translated to target instructions.

EMBODIMENT OF THE INVENTION

The preferred embodiment shows the techniques of this invention as applied to the emulation of the IBM S/390 (source) processor architecture on the 64-bit version of the IBM PowerPC (target) processor architecture. A 31-bit S/390 virtual or real address is assumed. A 24-bit ASID is assumed. A 4-bit S/390 PSW key is assumed. A maximum of 2 additional bits are allotted for other S/390 access control fields (see 08/349,771, which is assumed by this invention). A 64-bit PowerPC virtual address is assumed. A 32-bit PowerPC real address is assumed. The page size of both the S/390 and the PowerPC architectures is 4096 bytes. None of these metrics are essential to the proper operation of the invention. They represent specific attributes of this particular embodiment.

FIG. 3 illustrates the addressing relationship between a source instruction address in source exploded virtual form (**301**) and the virtual address of the ITM entry (**303**) that indicates its translation state. The low order zero of the exploded address indicates that source machine instructions must start on a two-byte boundary. The ASID and instruction address portion of the exploded address are shifted two bits left because each ITM entry is eight bytes wide (four times larger than the two byte logical instruction location). The base B is the starting point of the ITM area (**305**) in the target virtual address map (see FIG. 1). Concatenating B with the shifted ASID and instruction address gives the virtual address of the ITM entry for the instruction at the source exploded virtual address. As explained earlier, the ITM entry contains a target virtual address (**307**) to be branched to during source program execution. The address will be that of the target instructions performing the function of the source instruction (**306**) if a translation already exists. Box **304** represents the total virtual storage containing all source instructions in all address spaces. If a translation does not exist, the address will be that of the translator, which, when entered in this way, will resume instruction translation at the source exploded virtual address of the untranslated instruction of the ITM entry. In this embodiment, target branches must be to locations on 4-byte boundaries, so the two low-order bits of the address are used to indicate the span (**308**) of the translation in terms of two byte units. That is, since the source instructions of this embodiment can be of length 2, 4, or 6, the span can indicate the scope of a translation in terms of the source storage occupied by the instruction translated. This is used when stores into the instruction stream are detected in order to discover which translations are contaminated and must be invalidated. The first bit indicates whether or not the ITM entry represents two bytes that are part of an instruction starting before these two bytes, and the second bit specifies whether or not the next two bytes in source storage are part of the same instruction as these two bytes. This mechanism is general and allows specification for source instructions of any length. The values assumed in this embodiment are specifications of the assumed source and target processors and are not essential to the invention. Persons skilled in the art can easily design an ITM and addressing relationship to fit any variety of source instruction length and target branch address specification.

FIG. 4 illustrates the relationship between an ITM virtual section and a backing real ITM cache section. Generally, storage management efficiency would tend to cause an ITM section to be designed as a target page in size in a particular embodiment. In this embodiment that is 4K bytes. However, this is not key to the invention. As indicated, target Dynamic Address Translation (DAT) forms the link between a virtual ITM section and its real ITM backing section in real storage. The Target Virtual ITM Entry Address (**303**) is as described for FIG. 3. Target DAT results in the target real ITM Cache Entry Address (**402**) which accesses the ITM in target real storage. Box **305** represents the virtual storage of the virtual ITM, while box **404** represents the entire real storage allocation for the real ITM cache. Fields **307** and **308** represent the Translation pointer and Span in a virtual ITM entry, while **407** and **408** represent the same fields in the real ITM backing store. These are the very same fields with **307** and **308** in virtual storage, and **407** and **408** in real storage. The fields are 64 bits wide with two bits containing the instruction span information, as explained earlier.

FIG. 5 illustrates how the address of an existing backing real ITM section can be found, if one exists, when a page

fault occurs on a reference to an entry in a virtual ITM section. The process is described in FIG. 11. When such a page fault occurs, source DAT is performed on the ASID and instruction address (shown in FIG. 3) of the instruction to be performed in order to find its address in source real storage (504). In this embodiment, an ITM entry is 4 times as large as the instruction unit it represents. Therefore, if an ITM section size is chosen to be a page, the source storage section it represents is one-quarter of a page in size. Thus, to find an ITM section representing a particular instruction section it is necessary to include the two high-order bits of the within-a-page address of an instruction address, shown in the Source Real Storage Section Address (501) in FIG. 5 as XX, along with the Page-frame-real-address (PFRA). There is a Real ITM Directory (505) which indicates, for each Source Real Storage section, whether an ITM real section (509) exists in the Real ITM Cache or not, and, if so, what its address is. The Real ITM Directory Entry address (502) is calculated as shown by the relative source real storage section number (the PFRA extended on the right by the two high-order bits of the relative-within-a-page address, which are marked XX). In the general case, division is used, but in this embodiment shifting provides an efficient means for doing the division. In this embodiment, for example, an ITM section is 4K bytes, while a Real Storage section is 1K bytes. Thus, an ITM section represents the instructions in a quarter-page. To locate the proper ITM section for a source instruction, the relative address must include the two bits that specify which quarter of a page an instruction is in. This relative address is shifted to properly address the Directory entries as 4-byte entities. The base E is the starting point of the Directory in target storage, and is preappended to the relative Directory entry address to access it in target storage. As shown, the Real ITM Directory entry (508) contains the relative address of the Real ITM section for that source section, if one exists. It is combined with the base C to form the target machine real address (503) of the ITM section in the real ITM cache. It will be used to back all virtual ITM sections that represent those instructions. In other words, the same Real ITM section will back all virtual ITM sections for all address spaces that are in fact the same instructions. During emulation, this permits a set of the exact same instruction images which are a part of more than one instruction address space to use the same instruction translations. In S/390 systems, such storage areas, mapped into multiple address spaces, are called "Common" areas. For practicality, these areas are defined in units of the processor paging storage unit, i.e., they occupy an integral number of pages at page boundaries. In S/390, the same relative virtual pages of Common areas in all ASID's are backed in real storage by the very same pages since their content is, by definition, exactly the same. However, the exploded target virtual addresses of the areas are different in order to detect the specific ASID of reference in order to check the validity of the reference (see application Ser. No. 08/349,771, which is assumed by this invention). Since these different virtual addresses are really the "names" of the very same information, the same Real ITM section can back the different virtual ITM sections for them. If the Real ITM Directory indicates that there is no assigned Real ITM section in the Real ITM Cache, the emulator must newly assign a free page from the Real ITM Cache area to back the virtual ITM that is being processed, and indicate this in the Real ITM Directory Entry for the Source Real Storage section. This is described in FIG. 12.

FIG. 6 shows the relationship between source real storage and the target virtual ITR sections that contain the target translations of all source instructions in the source real storage sections. As shown on the left, a source page (607), addressed by a source exploded virtual address (601) and mapped into target exploded virtual storage as a contiguous page, is actually four contiguous storage sections. That is because, in this embodiment, an ITM entry is four times as wide as a source minimal logical instruction. To make an ITM section one target page, the source storage section covered by that ITM is one-quarter of a page. Performing source machine DAT on the source address portion of the source exploded virtual section address locates the source real section in real page 608 in source real storage. Since relocation occurs at the page level, the page is contiguous in target real storage within the portion reserved to represent source real storage. The source real storage section address in target real storage is obtained by pre-appending base D to the beginning of it. Each storage section of each source real page will have its source-to-target instruction translations placed into a different Instruction Translation Region (ITR) section (609), as indicated by the dotted line between the two rightmost rectangles in the figure. In this embodiment, the size of an ITR section has been specified as one megabyte, though that is not central to the invention. An ITR section should be large enough to be expected to hold all of the translations of a source real storage section in order to avoid the complications of storage management that would otherwise be required. Given a source real storage section address (501), shown in the top portion of the figure, the virtual address (603) of the corresponding ITR section can be calculated. However, ITR sections are backed in target real storage only as required to hold the translations of source instructions, and this assignment can occur by demand paging as required. Thus, the direct relationship of a virtual ITR section with each section of a source program that may contain instructions is not costly in terms of real resources required by such a virtual storage assignment. That is because target DAT is used for the translation of virtual ITR references to target real addresses, by executing the process in target machine DAT-on mode. The address translation indicated between the second and third address types illustrated at the top of the figure uses the base F to properly place the ITR address into the assigned part of target virtual reserved for ITR sections. The page number is shifted to accomodate the expansion of ITR required beyond the size of the storage section (to one megabyte multiples, one for each source storage section), and the two high-order bits of the within-page address are used to address the proper storage section within the page. Generally, this transformation is a multiplication operation, but in this specific embodiment shifting is used as an efficient multiplication means.

FIG. 7 shows the relationship between ITM entries in ITM section 509 in the Real ITM Cache (404) and areas of the single ITR section (609) assigned to hold instruction translations of source instructions in the real storage section whose translation state is indicated by the ITM entries (eight bytes wide). The real ITM cache section address is illustrated by 503. The virtual ITR section address is depicted by 603. The ITM page is shown at the bottom of the figure on the left. The virtual ITR section 609 within the virtual ITR region 606 addressed by these entries is shown on the bottom to the right. As shown, some entries address instruction translations. Others address the address of the Translator (or IST) found in each ITR header. The latter is true when no translation yet exists for a source instruction unit (two bytes in this embodiment) but an ITM page does exist in the Real ITM Cache (404) for the source section containing an instruction unit. The only address relationship between a

Real ITM Cache section and a Virtual ITR section is the direct address pointers within the ITM entries themselves.

An ITM section represents the instructions potentially in a target virtual exploded storage section. Many such sections can be represented by the same ITM section in real storage because they represent the same source byte images in source real storage. When it is found that a virtual ITM section represents a section of source real storage for which there is an already existing ITM section, the virtual ITM section for it is backed by the same existing ITM section in real storage. As shown in FIG. 5, a Real ITM Directory indicates which source sections have existing real ITM sections for them. Since two different target exploded virtual addresses can be backed by the same real target page frame, it is possible that two different target PTE's may define different virtual addresses backed by the same page frame. FIG. 8 indicates how this is kept track of in order to allow disallocation of an ITM page frame when source program execution makes that necessary. If an ITM page is to be invalidated, it is necessary to clear all PTE's defining target virtual addresses backed by the page in order to ensure correct operation. A Real ITM Cache Frame-to-PTE-List Directory, which has one entry per Real ITM page frame, contains the heads of pointer lists associating PTE entries addressing ITM entries backed by the same real page frame in the Real ITM Cache. A Single-Frame-PTE list entry is defined, one for each PTE entry, through which a list of associated PTE's can chain. In the figure, pointer **802**, in the real ITM cache frame-to-PTE list directory entry associated with real ITM page **801** in the real ITM cache (**810**) addresses the **803** entry in the PTE list (which is associated with PTE **805**). It in turn addresses the **804** entry in the list (associated with PTE **806**), which indicates the end of the list. This list is a consequence of having two PTE's, **805** and **806**, in the Target Page Table **809** defining virtual ITM addresses VA1 and VA2, both backed by the same real ITM page **801**. Should it be necessary to invalidate the ITM backed by page **801**, the list is used to invalidate all PTE's that address it. The general mechanism is described more fully in application Ser. No. 08/349,771, which is assumed by this invention.

FIG. **9** illustrates the target machine process for executing a source instruction. At step **901**, the source machine instruction address is preappended with the Address Space Identification (ASID) of the executing program and a base quantity offset for the beginning of the Instruction Translation Map (ITM) in target virtual storage, and shifted to accomodate for the width of an ITM entry (see FIG. **3**) and used to address the ITM entry for the source machine instruction to be executed. At step **902** the addressed ITM entry is loaded into a target machine working register. If a page fault results because of this fetch (**903**), the ITM is not backed in target real storage. The page fault handling is shown on FIG. **11**. If there is no page fault (and also following resolution of an ITM page fault event) the ITM entry is tested to find out if there is an existing target instruction translation for the current source instruction in step **904**. If not, a translation miss is said to have occurred. The translation miss processing is shown on FIG. **10**. If an instruction translation exists, the target instructions providing the effect of the source instructions being executed are themselves executed at step **905**. The source next-instruction address is then calculated at step **906** and control transfers to step **901** to perform the execution process for the next set of source instructions.

FIG. **10** illustrates the processing performed when a translation miss occurs. That is, there is no existing target machine instruction translation for the source machine instruction to be executed. Step **1001** determines the target machine virtual address of the Instruction Translation Region (ITR) section that will hold the instructions of the translation when they are produced. It is uniquely identified by the pointer to the translator entry in the ITR section, which pointer is found in the ITM entry which caused the translation miss, and was used to access this logic. (ITR sections are backed by target storage when accessed during processing but pages of ITR sections are pageable by the target real storage manager. Page faults on these pages are resolved by normal prior-art methods without special processing so the processing of such page faults is not shown here). The addressing relationship is depicted in FIG. **5**. Step **1002** calls the instruction translator (IST) to create the set of target instructions that will produce exactly the same effect in execution as the source instruction being executed, and to place them in the ITR section. On return from the translator, the address in the ITR of the newly created instruction translation is placed into the ITM entry for the source instruction, at step **1004**. For a source instruction which spans multiple instruction units, only the first ITM entry associated with the instruction is set to point to the translation. The instruction translation supplied by the translator will be used repeatedly each time the same source instruction is subsequently to be executed, assuming there has been no change to the source instruction in the meantime. The ITM indicates the location of the existing instruction translation allowing it to be directly invoked without further action by the translator function. Step **1005** indicates the span of the translation in the ITM entry to indicate the number of source bytes in the instruction stream covered by the translation in the ITR to which the ITM now points. This is done by marking each ITM entry for instruction units that are part of this same multi-unit source instruction as being associated with this source instruction as described in FIG. **3**. This can be used on stores into the instruction stream to indicate whether a translation has been invalidated by a store operation in the source program into an area that currently or previously contained instructions that were executed. The instruction translation is executed at step **1006**, the next source machine instruction address is calculated at step **1007**, and control transfers to the top of FIG. **9**, at step **901**, to perform the execution process for the next set of source instructions.

FIG. **11** shows the processing performed when a page fault occurs on the reference to an ITM entry in the processing depicted in FIG. **9**. The base portion of the extended target virtual machine address that caused the page fault is extracted at step **1101**. This is tested at step **1102** to check for an ITM page fault. If the base address portion is not that for the ITM table area in target virtual storage, some other type page fault has occurred and transfer is made to the real storage manager for routine page fault processing of the event. This is not shown here since many methods are previously described in prior art for demand paging of virtual storage. If step **1102** determines that the page fault was on an access to an ITM entry, step **1103** extracts the source ASID and source address, which indicate the ITM entry from the source exploded virtual address of the source instruction whose ITM entry is sought. For example, in S/390 ESA this would include the ASID and the virtual address of the instruction within that address space. Using the source machine Dynamic Address Translation (DAT) method, the source real address assigned to that source virtual address by the source operating system is calculated at step **1104**. Using the source real address, the address of the real ITM directory entry for the ITM section for the source

real storage page is calculated in step **1105**. There is a direct addressing relationship between source real storage pages and that directory of ITM sections in target real storage. The addressed directory entry is tested at step **1106** to ascertain whether or not an ITM section exists in real storage for that source real page. If not (the entry is NULL), control passes to FIG. **12** to handle the ITM miss condition. The initial state of the real ITM directory has all entries set to NULL. If the ITM section exists, a target machine Page Table Entry (PTE) is selected and allocated for this purpose at step **1107**, and the contents of the PTE are constructed to address the required page of ITM entries at step **1108**. At step **1109** store protection of the source page is established to allow detection by the target machine of store operations by target instructions of source machine translations into the page of source instructions represented by the ITM being established in the PTE. The PTE is made part of the single-frame PTE list for the target real page to which the PTE resolves references through it, at step **1110**. The structure of the single-frame PTE list is illustrated in FIG. **8**. Since multiple target virtual addresses of ITM section pages can resolve to the same ITM target real page frame address, a mechanism is needed to indicate all PTEs resolving to a single frame so that, if a frame is deallocated and reassigned to another use, all PTEs resolving to it can be invalidated. The single-frame PTE list is utilized to provide this function by relating all PTEs resolving to the same page frame real storage address. Control returns to step **904** of FIG. **9**.

FIG. **12** depicts the process for an ITM miss situation. If the ITM directory entry for an ITM section is NULL, real storage for an ITM cache section must be allocated. The storage is selected and allocated for the ITM section at step **1201** from the pool of target real pages assigned for this purpose. At step **1202** the target machine virtual address of the ITR section that is to hold the translations of source instructions from the source page represented by the ITM section being allocated in target real storage is computed. As explained earlier (see FIG. **5**), there is a direct addressing relationship between a source real page and a virtual ITR section whose address can be used to access the translations of source instructions in that page.

The source real page address computed in step **1104** of FIG. **11** is used to determine the virtual ITR section address.

Step **1203** calls a function that initializes the header of an ITR section in preparation for its receiving instruction translations. Step **1204** initializes the entries in the ITM section to transfer to the translator function, which will occur at source instruction execution until an entry is replaced by the target virtual address of a translation in the ITR section, by action of the translator, the first time the source instruction is encountered for execution. The format of an entry in the ITM section is depicted in FIG. **5**. Step **1205** sets the ITM directory entry for the real storage section (from **1104**) to address the newly allocated real ITM cache section. An entry in the real ITM directory is illustrated in FIG. **7**. All source exploded virtual storage pages represented by the newly-backed ITM section are store-protected in order to provide notification of any store that should cause invalidation of an existing source instruction translation, as was described in the Summary. This is done at step **1206**. Return is then made to step **1107** on FIG. **11** to continue ITM page fault processing.

FIG. **13** indicates the processing that occurs when a store protection exception is encountered during execution of the instructions of translations of source instructions, i.e., the source program is modifying an area that may contain instructions that may have already been translated to target instructions. Step **1301** calculates the source real address of the target virtual address that resulted in the protection exception. At step **1302**, the source real address of the reference is used to determine the real ITM directory entry for the ITM for instructions in the source page of the reference. Step **1303** tests that real ITM directory entry. If it is NULL, no instruction translations exist for that source storage section. The protection violation may be a source architecture violation. The rules of the source architecture are used to check for such a violation at step **1304**. If there has been such a violation, it is reported to the source operating system under the source architecture rules for handling such notification within the source operating environment. If not, the store protection is removed at step **1305**, the store is performed in the source storage within target storage at step **1306**, and return is made to the instruction following that which caused the exception in the target translation of the subject instruction specifying the store operation. If the real ITM directory entry was found not to be NULL at step **1303**, a check is made using source architecture rules to ascertain whether or not a source protection exception occurred at step **1307**. If so, this is communicated to the source operating system in accordance with the source rules for source communication of such an event in the source machine. If it is not a source program violation, then step **1308** tests for a translation corruption, i.e, whether or not the store changes a source location for which there exists a current target machine translation. The storage extent of the store operation is calculated and that extent is checked in the corresponding ITM section to see whether the change is affecting any part of the span of any translated source instruction(s). If not, the store is performed at step **1309** in the source storage as mapped in target storage, and control returns to the next target instruction to be executed within a source instruction translation. If there has been an instruction translation corruption as a result of the store operation in source machine storage, the translation management function is called to invalidate those translations affected, at step **1310**. Control then passes to step **1309** to perform the specified store operation in source machine real storage.

Finally, it is necessary to extend the PTE allocation process described in FIG. **27** of application Ser. No. 08/349,771, which is assumed by this invention. When a new PTE is allocated to back a target exploded virtual address with a target real page frame which represents a source real page, the real ITM directory described in this invention must be consulted to determine if the real page has an real ITM cache section allocated to it. If so, the PTE must be store protected to trap stores into a page which contains source instructions.

While the invention has been described in detail herein in accordance with certain embodiments thereof, many modifications and changes therein may be effected by those skilled in the art. Accordingly, it is intended by the appended claims to cover all such modifications and changes as fall within the true spirit and scope of the invention.

Having thus described our invention, what we claim as new and desire to secure by Letters patent is:

1. An emulation method for executing individual source instructions in a target processor to execute source programs requiring source processor features not built into the target processor, comprising the steps of:

inputting instructions of a source processor program to an emulation target processor having significant excess virtual addressing capacity compared to a virtual addressing capacity required for a source processor to natively execute the source processor program, and

supporting multiple source virtual address spaces in the operation of the source processor,

building a virtual ITM (instruction translation map) in a target virtual address space supported by the target processor, the virtual ITM containing an ITM entry for each source instruction addressable unit, each source instruction addressable unit beginning on a source storage instruction boundary, structuring each ITM entry for containing a translation address to a target translation program that executes a source instruction having a source address associated with the ITM entry,

determining a ratio R by dividing the length of each ITM entry by the length of each source instruction addressable unit,

accessing an ITM entry for an executing source instruction by:

generating a source aggregate virtual address for the source instruction by combining the source address of the source instruction with a source address space identifier of a source virtual address space containing the instruction,

multiplying the source aggregate virtual address by R to obtain a target virtual address component, and

inserting the target virtual address component into a predetermined component location in a target virtual address to generate an ITM entry target virtual address for locating an ITM entry associated with the source instruction in order to obtain a one-to-one addressing relationship between ITM entry target virtual addresses and source instruction addresses.

2. A source emulation system for executing source instructions in a target processor as defined in claim 1, further comprising the step of:

structuring each ITM entry with at least two fields: a pointer field for containing a target instruction translation address, and a span field for containing a span indication that indicates if the ITM entry is part of a set of ITM entries associated with the same source instruction for handling source variable length instructions.

3. A source emulation system for executing source instructions in a target processor as defined in claim 2, further comprising the step of:

using the source real address of a source instruction represented by an ITM entry to determine if an ITM section exists for the virtual ITM section containing the entry, and

translating an ITM entry target virtual address to a backing ITM entry target real address of a backing ITM entry in an assigned target page frame (PF) in real storage of a target processor, the target PF containing a subset of backing ITM entries for backing a section of the virtual ITM in target real storage, each backing ITM entry in the target PF supporting the pointer field and the span field in a corresponding virtual ITM entry.

4. A source emulation system for executing source instructions in a target processor as defined in claim 3, further comprising the step of:

signalling a page fault if the ITM section target virtual address has no backing target PF,

using source DAT to determine the source real address of the currently executing instruction,

using the source real address of the instruction to index to an entry in the real ITM directory to determine if a real ITM cache section is already defined for the source real storage section which includes the currently executing source instruction,

allocating a target PF from the real ITM cache if one does not exist,

initializing the target PF when it is assigned to back the ITM target virtual address by setting each pointer field in the target PF to a value interpreted as a translator address for locating a translator program in the target virtual storage for generating target translations which are target programs that execute source instructions, and

indicating the target PF in the real ITM directory entry associated with the source real storage section containing the currently executing source instruction.

5. A source emulation system for executing source instructions in a target processor as defined in claim 4, further comprising the step of:

signalling a page fault if the ITM section target virtual address has no backing target PF,

using source DAT to determine the source real address of the currently executing instruction,

using the source real address of the instruction to index to an entry in the real ITM directory to determine if a real ITM cache section is already defined for the source real storage section which includes the currently executing source instruction, and

backing the virtual section ITM section for the currently executing source instruction with the same real ITM cache section if one exists.

6. A source emulation system for executing source instructions in a target processor as defined in claim 5, further comprising the steps of:

accessing the pointer field of the backing ITM entry for obtaining a translator address to enter the translator program,

assigning a virtual instruction translation region (virtual ITR) in target virtual storage for locating the translator program and instruction translations of previously executed source instructions,

executing the translator program for generating a target translation for a currently executing source instruction when the backing ITM entry does not contain a pointer to an instruction translation,

storing the target translation in a page within a virtual ITR section, associated with the source section containing the instruction and located by a translation target virtual address,

writing the translation target virtual address in the pointer field of the backing ITM entry for the executing source instruction to overlay the translator address to enable a direct access of the target translation from the backing ITM entry for enabling a later execution of the source instruction to bypass the translator program, and

setting the span field in the backing ITM entry to a span indication indicating if the ITM entry is part of a set of consecutive backing ITM entries associated with the executing source instruction, and setting the span field in each other backing ITM entry in the set to a span indication that indicates which backing ITM entries are part of the consecutive set.

7. A source emulation system for an executing source instruction in a target processor as defined in claim 6, further comprising the steps of:

accessing the translation target virtual address in the pointer field in the backing ITM entry when subsequently executing a previously executed source instruction associated with the ITM entry,

performing dynamic address translation (DAT) on the translation target virtual address to locate a backing target PF in target real storage containing at least the beginning of the target translation for the previously executed source instruction, and

executing target instructions of the target translation in the target PF to emulate execution of the executing source instruction.

8. A source emulation system for executing source instructions in a target processor as defined in claim 7, further comprising the step of:

signalling an ITR page fault if the translation target virtual address has no backing target PF for the ITR page.

9. An emulation system for an executing source instructions in a target processor as defined in claim 2, further comprising the step of:

accessing the backing ITM entry in the backing section target PF at the target real address associated with the source instruction located at a different source virtual address location by using a source DAT process, wherein the backing section of the target PF is used to back multiple copies of the same instruction in different source virtual locations to avoid having multiple copies of the backing section in target real storage for multiple source copies of the same source instruction.

10. A source emulation system for executing source instructions in a target processor as defined in claim 6, further comprising the steps of:

determining if an existing ITR virtual pointer in an existing real ITM entry in a backing PF locates an existing instruction translation previously determined for the same source instruction as the currently executing source instruction but at a different source virtual address, and

backing the ITM section for the currently executing source instruction by the same PF, and the same instruction translation being used for multiple source virtual addresses containing the same source instruction.

11. A source emulation system for executing source instructions in a target processor as defined in claim 7, further comprising the steps of:

protecting target page frames (PFs) in target storage which contain previously translated source instructions from being stored into by the target processor,

detecting each source real storage address of each source instruction subject to a source store operation,

determining all target virtual addresses of all ITM entries associated with previously translated source instructions detected as being stored into,

using the span field of the ITM entry associated with the first source byte store into to determine the first source byte of the first source instruction unit affected by a store operation,

setting the ITM entry for that first affected source instruction to an untranslated state,

setting to an untranslated state the pointer field in a first ITM entry associated with the first instruction unit of each other translated instruction affected by a store operation,

also setting the span field to an unassociated state in all ITM entries associated with each source instruction stored into, the settings in the pointer and span fields of all ITM entries associated with all stored-into source instructions being set to indicate no association with any source instruction, and

then performing corresponding store operations in target real storage for the same instructions.

12. A source emulation system for executing source instructions in a target processor as defined in claim 7, further comprising the step of:

maintaining a fixed allocation relationship of ITR virtual sections to source real storage sections, using the fixed allocation relationship to directly access an ITR virtual section given a source real storage section address, and using the real ITR pages as needed to back virtual ITR pages to hold instruction translations for efficiently using real storage, while using target virtual storage to provide direct addressing access of the ITR pages.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,560,013

DATED : Sep. 24, 1996

INVENTOR(S) : Casper A. Scalzi et al.

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Column 6, line 44 **"2**(46)=65536" should be --2**16 = 65536--.**

Signed and Sealed this

Seventh Day of January, 1997

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer *Commissioner of Patents and Trademarks*

US006026240A

United States Patent [19]

Subramanian

[11] **Patent Number: 6,026,240**

[45] **Date of Patent: Feb. 15, 2000**

[54] **METHOD AND APPARATUS FOR OPTIMIZING PROGRAM LOOPS CONTAINING OMEGA-INVARIANT STATEMENTS**

[75] Inventor: **Krishna Subramanian**, Mountain View, Calif.

[73] Assignee: **Sun Microsystems, Inc.**, Palo Alto, Calif.

[21] Appl. No.: **08/609,035**

[22] Filed: **Feb. 29, 1996**

[51] **Int. Cl.**^{7} **G06F 9/45**

[52] **U.S. Cl.** **395/709**; 395/705

[58] **Field of Search** 395/702, 705, 395/709

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,202,995	4/1993	O'Brien	395/709
5,361,354	11/1994	Greyzck	395/705
5,457,799	10/1995	Srivastava	395/705
5,596,732	1/1997	Hosoi	395/705

OTHER PUBLICATIONS

Zima, Hans, and Barbara Chapman, Supercompilers for Parallel and Vector Computers, New York, NY, ACM Press pp. 112–172, 1991.

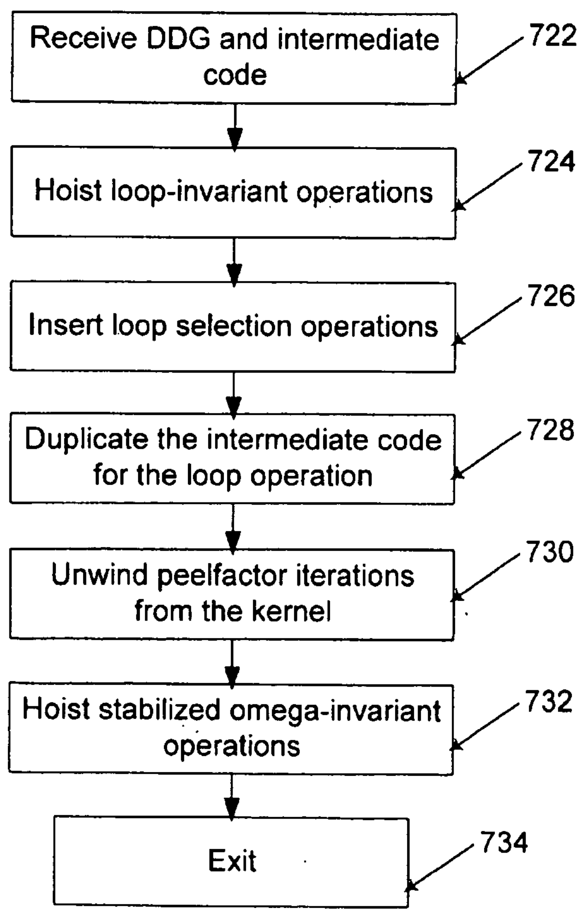
Aho, Alfred V., et al., *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison Wesley (1986), pp. 638–642.

Primary Examiner—Alvin E. Oberley
Assistant Examiner—Peter Stecher
Attorney, Agent, or Firm—Beyer & Weaver, LLP

[57] **ABSTRACT**

Apparatus, methods, and computer program products are disclosed for optimizing programs containing single basic block natural loops with a determinable number of iterations. The invention optimizes, for execution speed, such program loops containing statements that are initially variant, but stabilize and become invariant after some number of iterations of the loop. The invention optimizes the loop by unwinding iterations from the loop for which the statements are variant, and by hoisting the stabilized statement from subsequent iterations of the loop.

10 Claims, 9 Drawing Sheets



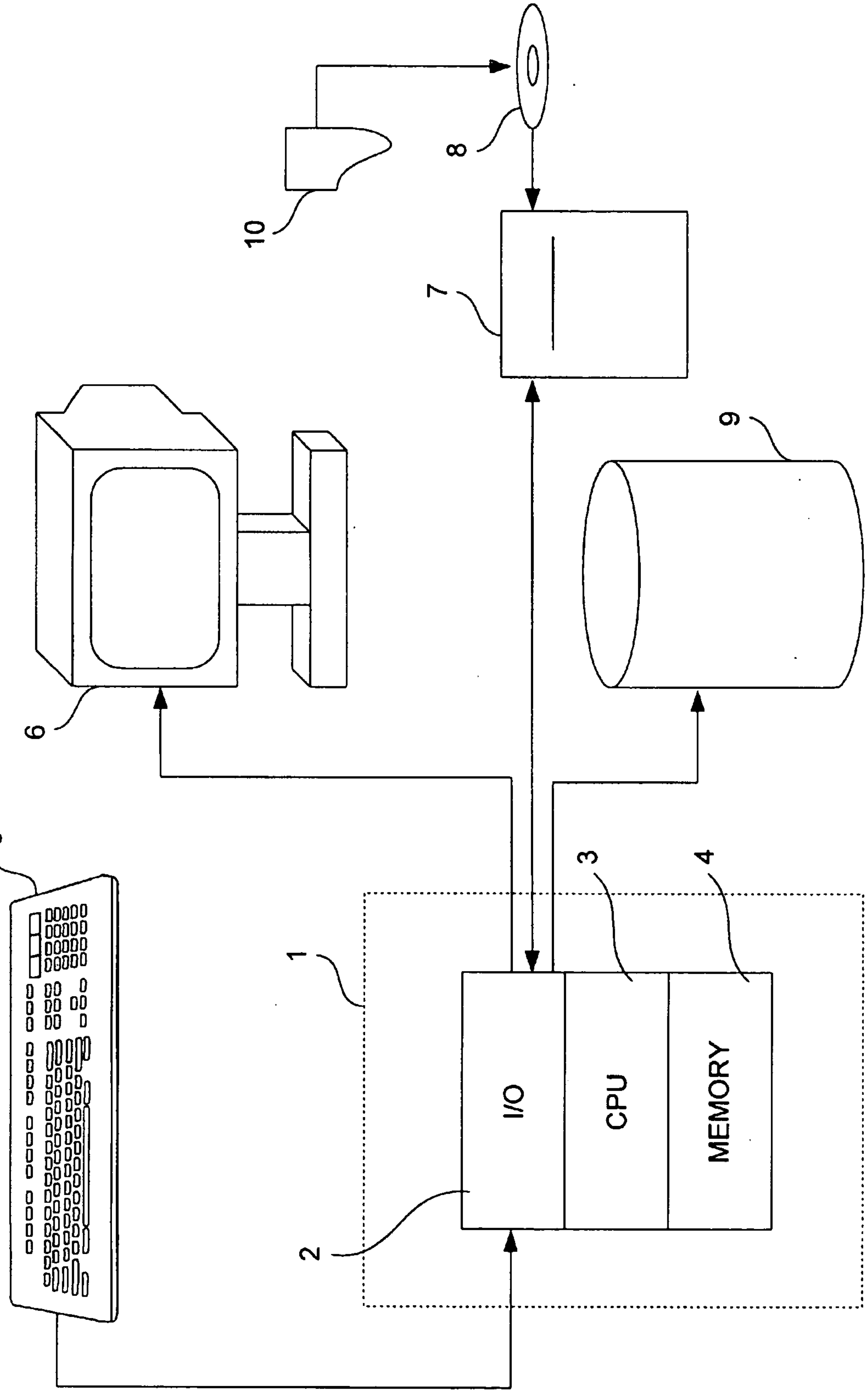


FIG. 1

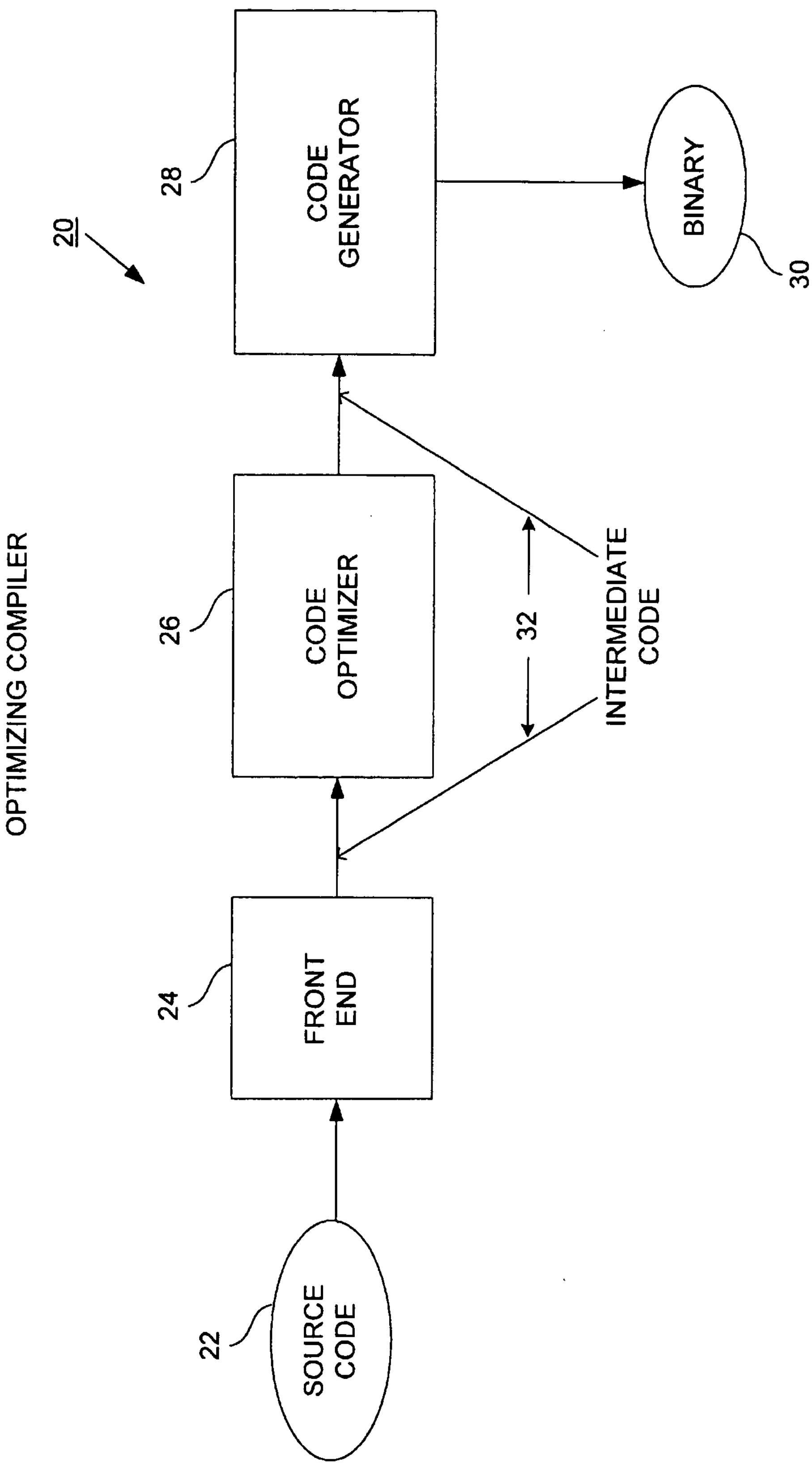
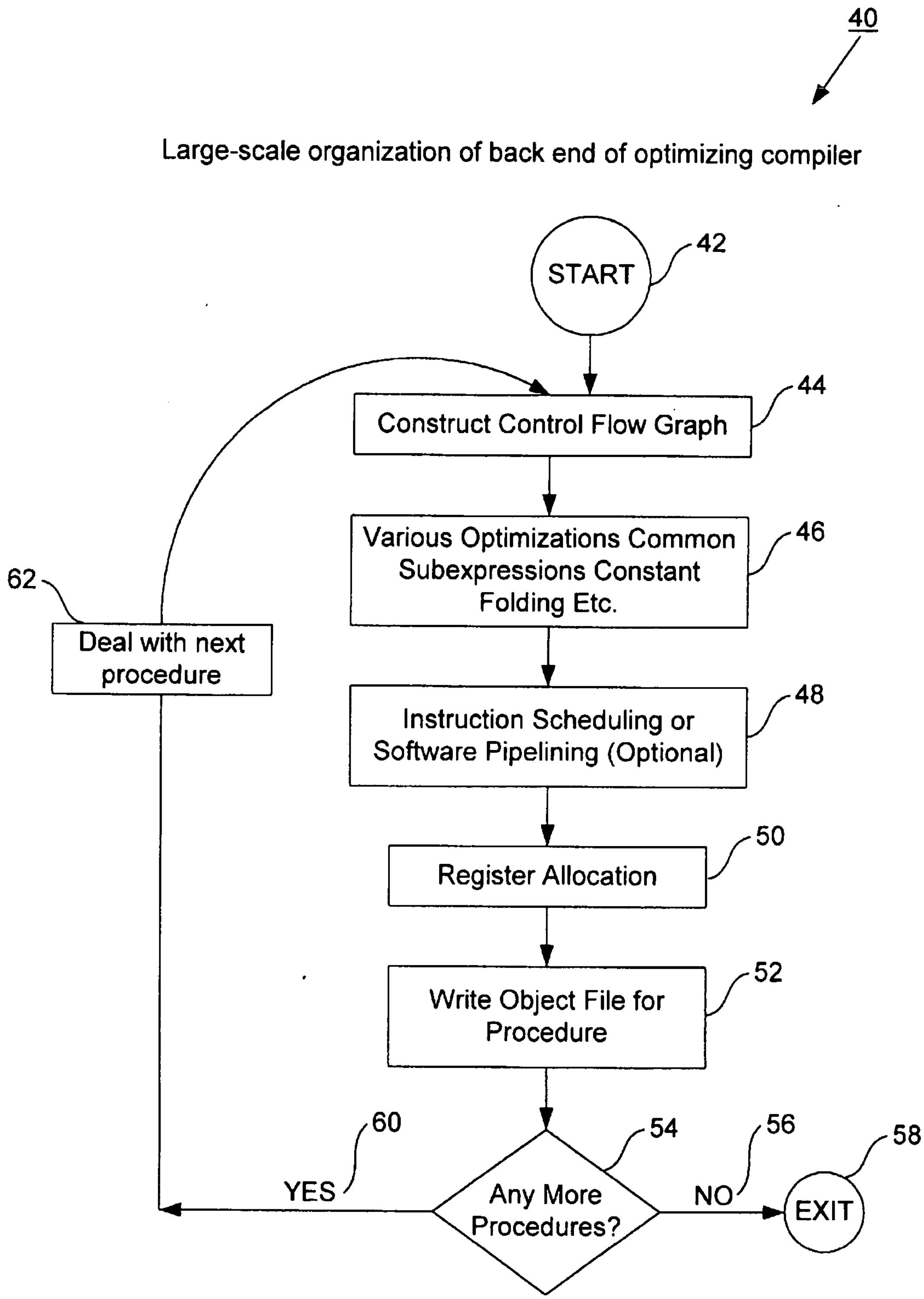

OPTIMIZING COMPILER
20
22
SOURCE CODE
24
FRONT END
26
CODE OPTIMIZER
28
CODE GENERATOR
BINARY
30
32
INTERMEDIATE CODE

FIG. 2



40
Large-scale organization of back end of optimizing compiler
START
42
Construct Control Flow Graph
44
Various Optimizations Common Subexpressions Constant Folding Etc.
46
Instruction Scheduling or Software Pipelining (Optional)
48
Register Allocation
50
Write Object File for Procedure
52
Any More Procedures?
54
NO
56
EXIT
58
YES
60
Deal with next procedure
62

FIG. 3

```
For  ( i=0; i<=N; i++) {
        z = x - 2;        //node 1
        x = a + b;        //node 2          401
        y = x - 1;        //node 3
}
```

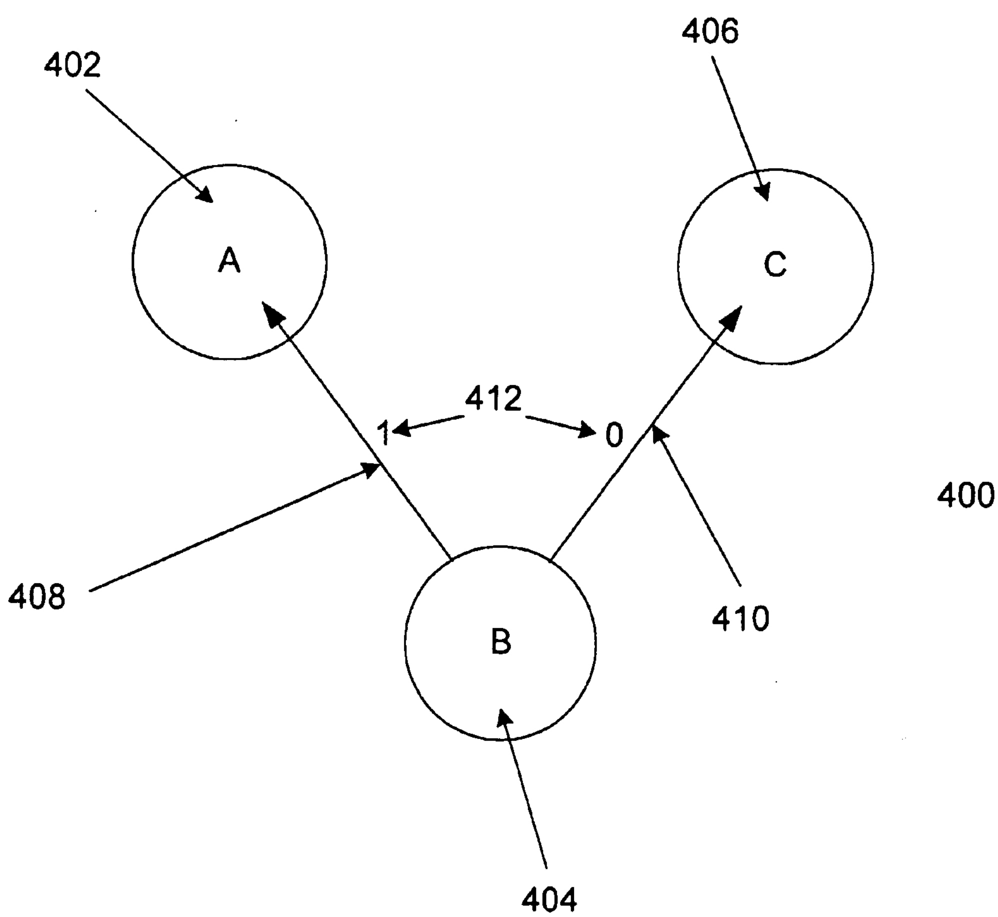


FIG. 4

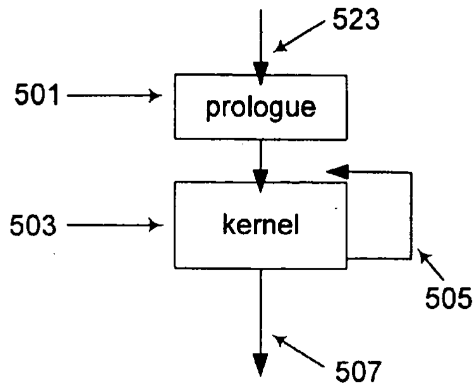

523
501
prologue
503
kernel
505
507

FIG. 5A

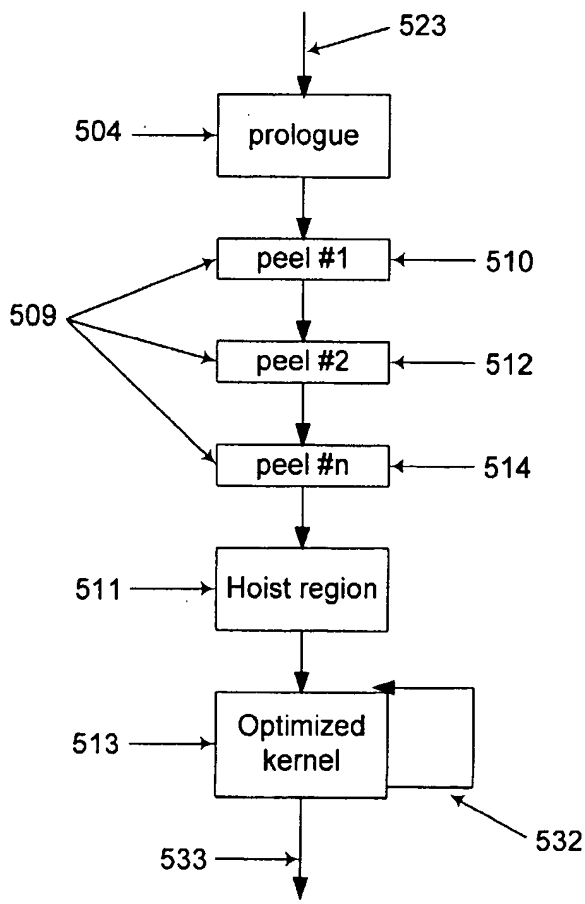

523
504
prologue
peel #1
510
509
peel #2
512
peel #n
514
511
Hoist region
513
Optimized kernel
532
533

FIG. 5B

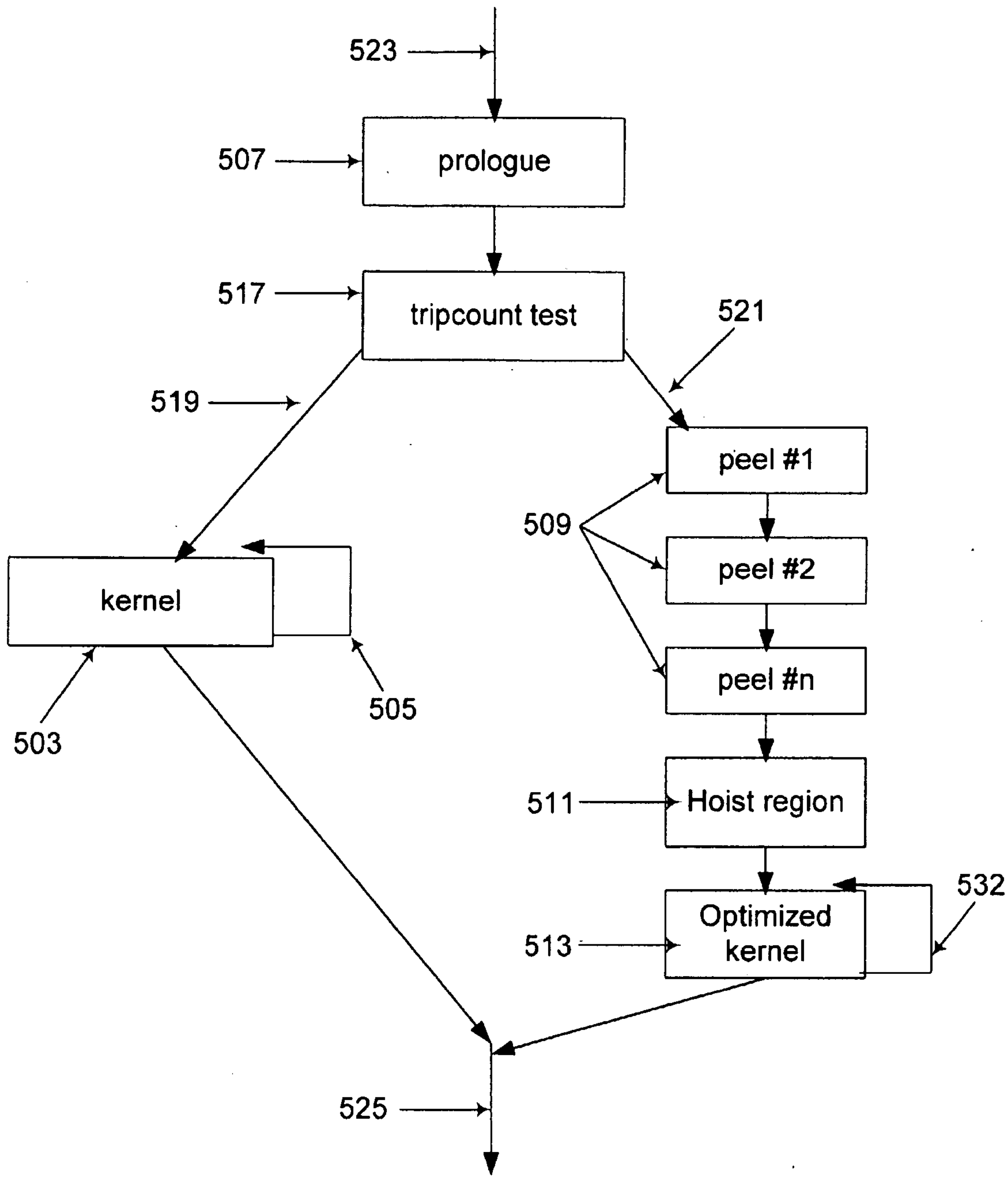
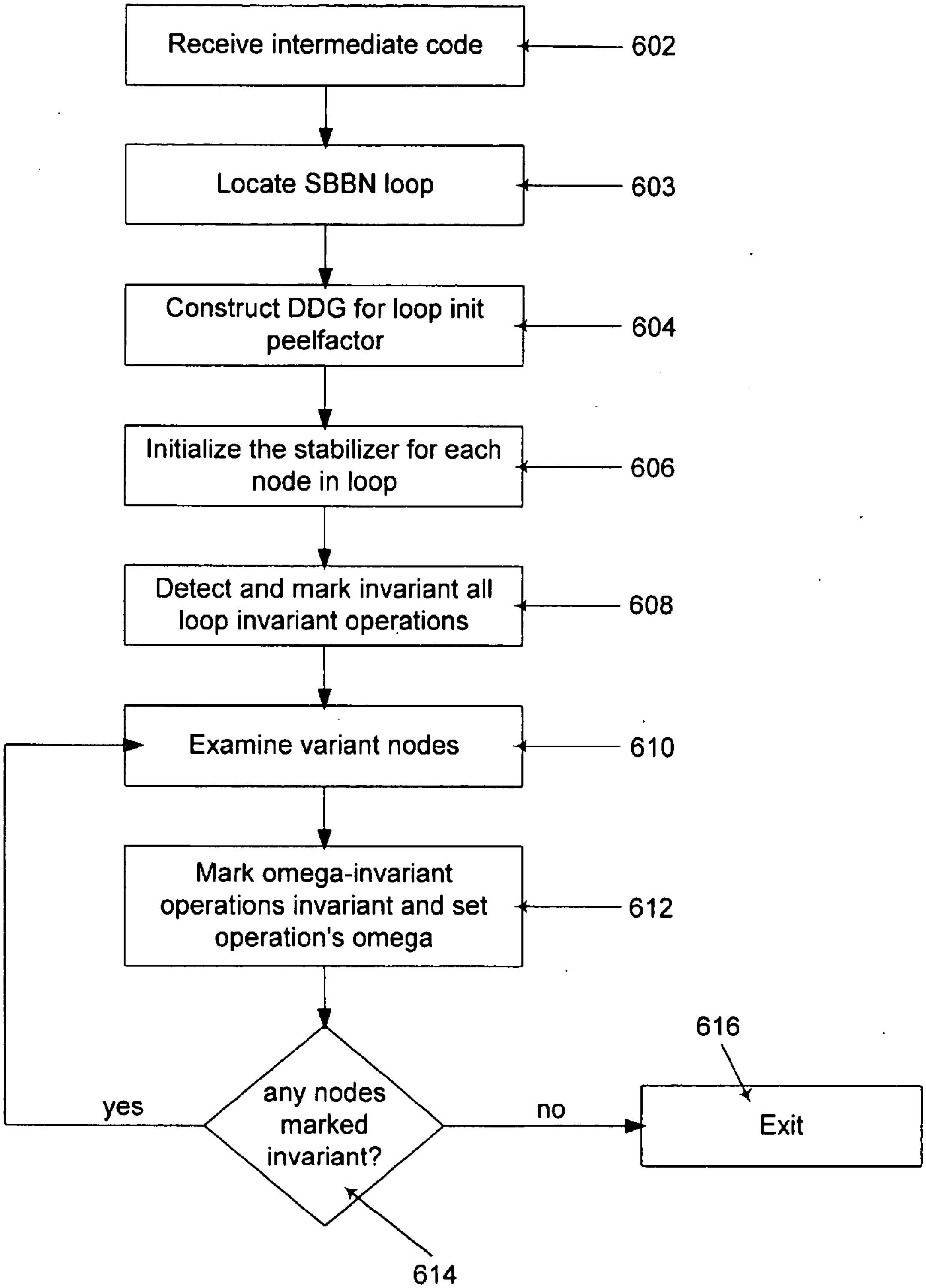

523
507
prologue
517
tripcount test
521
519
peel #1
509
peel #2
kernel
peel #n
503
505
511
Hoist region
532
Optimized kernel
513
525

FIG. 5C



Receive intermediate code
602
Locate SBBN loop
603
Construct DDG for loop init peelfactor
604
Initialize the stabilizer for each node in loop
606
Detect and mark invariant all loop invariant operations
608
Examine variant nodes
610
Mark omega-invariant operations invariant and set operation's omega
612
616
yes
any nodes marked invariant?
no
Exit
614

FIG. 6

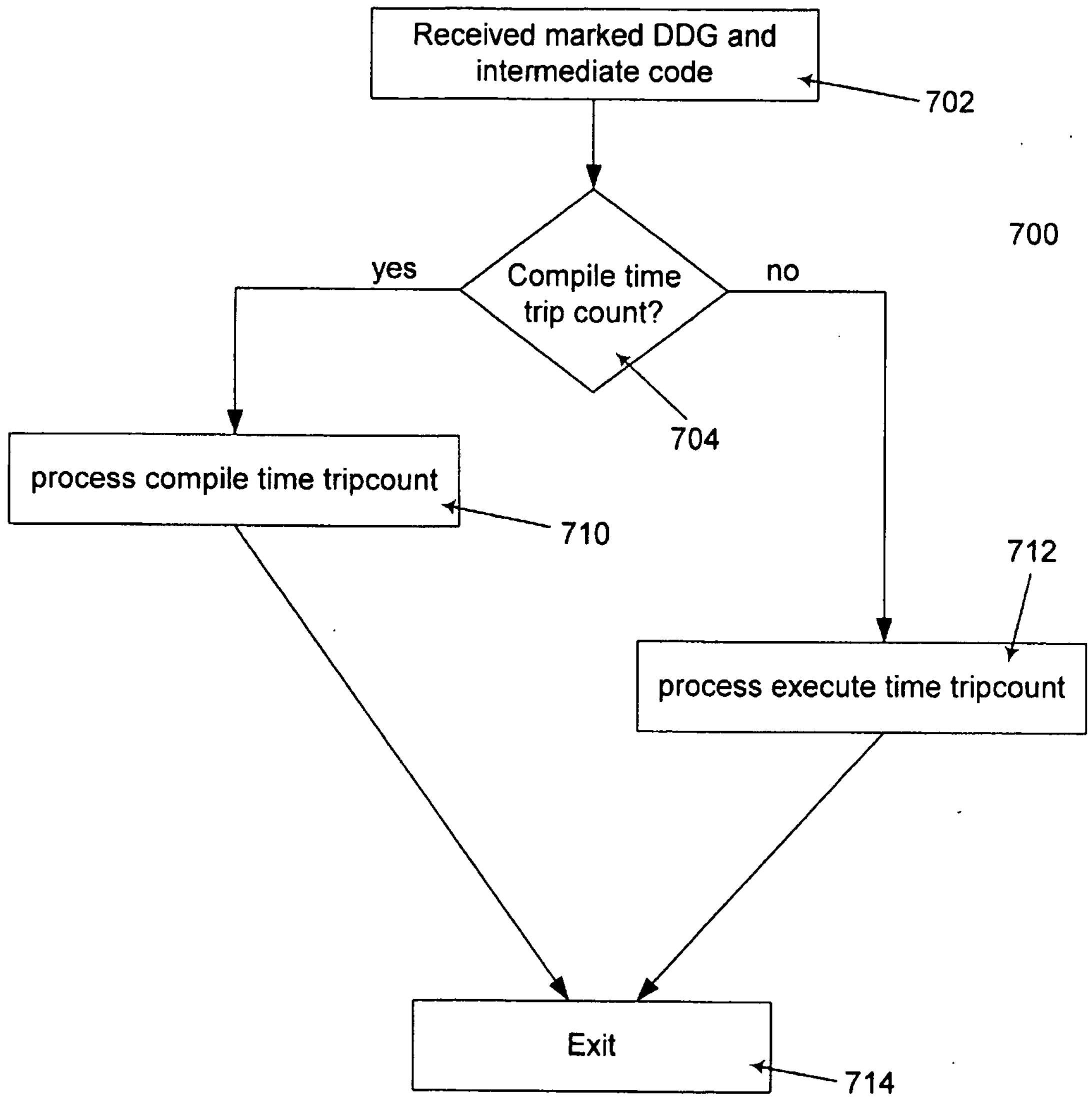

Received marked DDG and intermediate code
702
700
yes
Compile time trip count?
no
704
process compile time tripcount
710
712
process execute time tripcount
Exit
714

FIG. 7

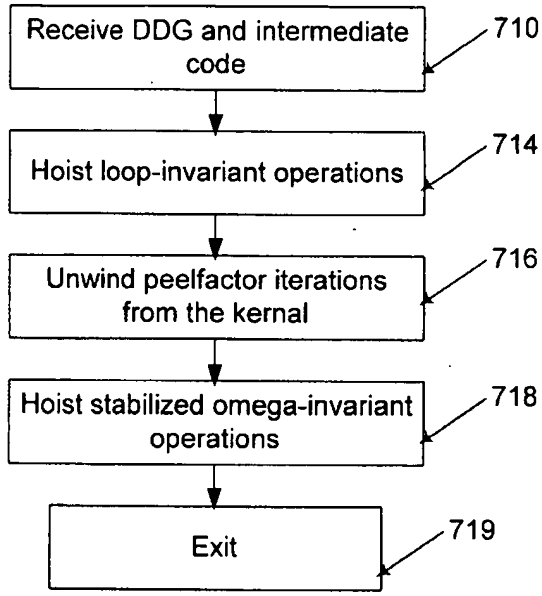

Receive DDG and intermediate code
710
Hoist loop-invariant operations
714
Unwind peelfactor iterations from the kernal
716
Hoist stabilized omega-invariant operations
718
Exit
719

FIG. 7A

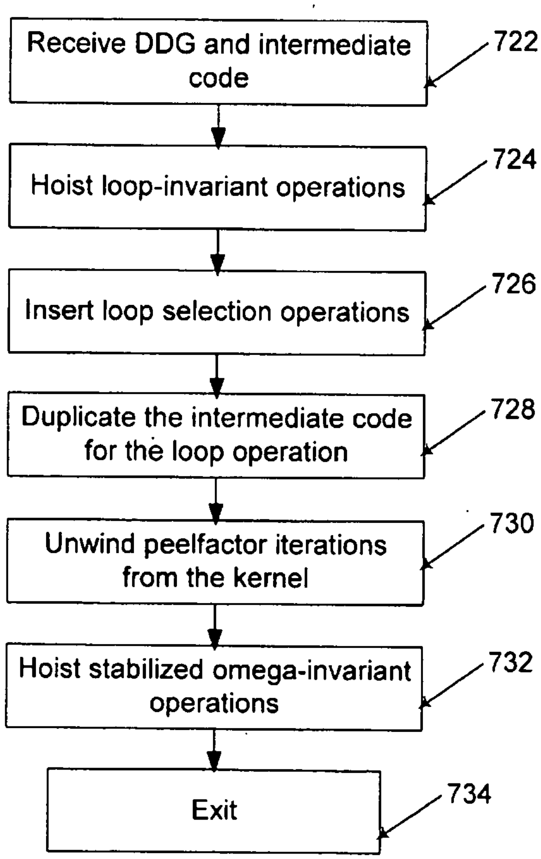

Receive DDG and intermediate code
722
Hoist loop-invariant operations
724
Insert loop selection operations
726
Duplicate the intermediate code for the loop operation
728
Unwind peelfactor iterations from the kernel
730
Hoist stabilized omega-invariant operations
732
Exit
734

FIG. 7B

METHOD AND APPARATUS FOR OPTIMIZING PROGRAM LOOPS CONTAINING OMEGA-INVARIANT STATEMENTS

FIELD OF THE INVENTION

This invention relates to the field of Optimizing Compilers for computer systems. Specifically, this invention is a new and useful optimization method and apparatus for optimizing the order of computer operation codes resulting from the compilation of a program loop.

BACKGROUND

Early computers were programmed by rewiling them. Modern computers are programmed by arranging a sequence of bits in the computer's memory. These bits perform a similar (but much more useful) function as the wiring in early computers. Thus, a modern computer operates according to the binary instructions resident in the computer's memory. These binary instructions are termed operation codes (opcodes). The computer fetches an opcode from the memory location pointed to by a program counter. The computer's central processor unit (CPU) evaluates the opcode and performs the particular operation associated with that opcode. Directly loading binary values in memory to program a computer is both time consuming and mind numbing. Programming languages simplify this problem by enabling a programmer to use a symbolic textual representation (the source code) of the operations that the computer is to perform. This symbolic representation is converted into binary opcodes by compilers or assemblers. By processing the source code, compilers and assemblers create an object file containing the opcodes corresponding to the source code. This object file, when linked to others, results in executable instructions that can be loaded into a computer's memory and executed by the computer.

A source program consists of an ordered grouping of strings (statements) that are converted into a binary representation (including both opcodes and data) suitable for execution by a target computer architecture. A source program provides a symbolic description of the operations that a computer will perform when executing the binary instructions resulting from compilation and linking of the source. The conversion from source to binary is performed according to the grammatical and syntactical rules of the programming language used to write the source. This conversion from source to binary is performed by both compilers and assemblers.

One significant difference between assemblers and compilers is that assemblers translate source code statements into binary opcodes in a one-to-one fashion (although some "macro" capabilities are often provided). On the other hand, compilers transform source code statements into sequences of binary opcodes that, when executed in a computer, perform the operation described by the source. The symbolic statements processed by a compiler are more general than those processed by an assembler and each compiled statement can produce a multitude of opcodes that, when executed by a computer, implement the operation described by the symbolic statement. Unlike an assembler, that maintains the essential structural organization of the source code when producing binary opcode sequences, a compiler may significantly change the structural organization represented by the source when producing the compiled binary. However, no matter how much the compiler changes this organization, the compiler is restricted in that the compiled binary, when executed by a computer, must provide the same result as the programmer described using the source language—regardless of how this result is obtained.

Many modern compilers can optimize the binary opcodes resulting from the compilation process. Due to the design of programming languages, a compiler can determine structural information about the program being compiled. This information can be used by the compiler to generate different versions of the sequence of opcodes that perform the same operation. (For example, enabling debugging capability, or optimizing instructions dependant on what version of the target processor the source code is compiled for.) Some optimizations minimize the amount of memory required to hold the instructions, other optimizations reduce the time required to execute the instructions. The invention disclosed herein optimizes so as to maximize execution speed for a particular type of loop operation.

Some advantages of optimization are that the optimizing compiler frees the programmer from the time consuming task of manually tuning the source code. This increases programmer productivity. Optimizing compilers also encourage a programmer to write maintainable code because manual tuning often makes the source code less understandable to other programmers. Finally, an optimizing compiler improves portability of code because source code tuned to one computer architecture may be inefficient on another computer architecture.

Compilers generally have three segments: (1) a front-end that processes the syntax and semantics of the language and generates at least one version of an "intermediate" code representation of the source; (2) a back-end that converts the intermediate code representation into binary computer instructions (opcodes) for a particular computer architecture (i.e., SPARC, X86, IBM, etc.); and (3) various code optimization segments between the front- and back-ends of the compiler. These optimization segments operate on, and adjust, the intermediate code representation of the source. For loops, the intermediate code representation generally includes data structures that either represent, or can be used to create, data dependency graphs (DDGs). DDGs embody the information required for an optimizer to determine which statements are dependent on other statements. The nodes in the graph represent statements in the loop and arcs represent the data dependencies between nodes. Data dependency graphs are described in chapter 4 of *Superconpilers for Parallel and Vector Computers*, by Hans Zima, ACM press, ISBN 0-201-17560-6, 1991.

One example of a prior art optimization is for the compiler to process the source code as if the programmer had written the source in a more efficient manner. For example, common subexpression elimination replaces subexpressions that are used more than once with a temporary variable set to the subexpression's value. Thus:

```
a=i*2+3;
b=sqrt(i*2);
```

compiles as if written as:

```
temp=i*2;
a=temp +3;
b=sqrt(temp);
```

Another optimization is by code motion. This optimization hoists, from out of the enclosing loop, expressions that are loop-invariant for each iteration. Thus:

```
while (!feop(fp)) DoSomething (fp, x*5);
```

compiles as if written as:

```
temp=x*5;
```

while (!feof(fp)) DoSomething (fp, temp);

Yet another optimization (trading memory for speed) is to expand the expressions contained in the loop so that less time is spent performing loop overhead operations. For example:

a=0;

for (i=0; i<100; i++) a=a+i;

can be compiled as if written as:

```
a = 0
for (i = 0; i < 100; i + = 5) {
    a = a + i;
    a = a + i + 1;
    a = a + i + 2;
    a = a + i + 3;
    a = a + i + 4;
}
```

Finally, the compiler could just unwind iterations from the loop so that there would be no loop overheads. Thus,

a=0;

for (i=1; i<6; i++) a=a+i;

can be compiled as if written as:

```
a = 0;
{
    a = a + 1;
    a = a + 2;
    a = a + 3;
    a = a + 4;
    a = a + 5;
}
```

In the above examples, the plus "+" is used to indicate any language operator. Further, there are other optimizations that could be performed on each example. The above examples indicates the operation of each separate optimization. In an optimizing compiler, many optimizations are applied to best optimize the resultant executable instructions.

Of course, the original source code is not modified by the compiler, rather the compiler sufficiently understands the program structure, as described by the source, to make these optimizations for the programmer without changing the programmer's intended result. These optimizations result in the production of binary instructions that execute faster in the target computer than the non-optimized instructions that perform the same operations.

It is well understood in the art how to hoist loop-invariant operations out of a loop operation. A general discussion of optimizing compilers and the related prior art techniques can be found in *Conipilers: Principles, Techniques and Tools* by Alfred V. Aho, Ravi Sethi and Jeffery D. Ullman, Addison-Wesley Publishing Co., 1988, ISBN 0-201-10088-6, hereinafter Aho. Optimization of loop-invariant computations is described in Aho at pages 638–642.

However, it is not known to the art how to hoist certain statements that are almost loop-invariant. In particular, some compiler source sequences are not initially loop invariant, but become invariant after a given number of iterations. For example:

```
x = 1;
for (i = 0; i < = 10; i++) {
  y = x * 5;      //inst 1
  x = 2;          //inst 2
}
```

Here, on the first iteration of the loop, y receives the value of (1*5=5), and x receives the value 2. During the second iteration, y receives the value (2*5=10), and x again receives value 2. On the third iteration and continuing until the end of the loop, y receives the same value (10). In this example, the second statement of the loop "x=2; " is not loop-invariant because the value of x that reaches "inst 1" in the first iteration is different than the value of x that reaches "instr 1" in subsequent iterations. Thus, the prior alt cannot hoist the operations representing this statement from the loop's kernel. The same applies to variable y. However, x does not change subsequent to the first iteration over the loop. Thus, x and y become invariant after the first iteration. Keeping the operations represented by these statements within the loop's kernel, after the first iteration, slows execution of the loop because each execution of an instruction takes time and computer resources. After the initial variant iterations of the loop, these statements have no further utility. Thus the time spent processing these statements is wasted and the loop is correspondingly inefficient.

Statements that have assignments that are variant in the first omega iterations, but that become invariant after omega iterations are termed "omega-invariant". Omega-invariant statements "stabilize" after "omega" iterations. Hence, the value delivered by the statement stabilizes after omega iterations and becomes invariant for subsequent iterations. In the example above, the omega for "inst 1" is one because the value of y remains the same (10) after one iteration of the loop. The omega for "instr 2" is zero because it is assigned the value of 2 in the first and all subsequent iterations of the loop.

The prior art does not optimize this sequence of statements. Thus, prior art compilers generate less efficient (slower) loops than compilers practicing the invention. The invention described herein optimizes the execution speed of SBBN loops containing omega-invariant statements with a determinable number of iterations.

SUMMARY OF THE INVENTION

The present invention overcomes the disadvantages of the above described systems and has the specific utility, among others, of providing an economical, high performance, adaptable system, apparatus, method, system and computer program product for increasing the execution speed of target programs by reducing the number of instructions repeatedly executed during some programmed loops. One aspect of the present invention identifies program loop statements that contain omega-invariant statements and optimizes the execution speed of these loops.

In another aspect of the invention, a computer system is disclosed that has a central processing unit (CPU) and random access memory (RAM) coupled to the CPU, for use in compiling a target program to run on a target computer architecture. In addition, this system has an optimizing compiler capable of identifying single-basic-block-natural loops with a determinable tripcount and containing an omega-invariant operation. Further, this compiler is capable of detecting the contained omega-invariant statement and of determining the number of iterations required to stabilize the

omega-invariant statement. Finally, the compiler is capable of optimizing the loop for increased execution speed dependent upon the number of iterations required to stabilize the omega-invariant statement.

In one aspect of the invention, an apparatus is disclosed using a computer and compiler system to optimize single-basic-block loops containing omega-invariant statements that stabilize after a determinable number of iterations.

In another aspect of the invention a code optimizer for use in a compiler operating on a computer system is disclosed for detecting and optimizing single-basic-block loops containing omega-invariant statements.

In yet another aspect of the invention, a method is disclosed for identifying and optimizing single-basic-block loops containing omega-invariant statements.

Finally, another aspect of the invention discloses a computer program product to cause a computer to detect and optimize single-basic-block loops containing omega-invariant statements.

DESCRIPTION OF THE DRAWINGS

The objects, features and advantages of the system of the present invention will be apparent from the following description in which:

FIG. 1 illustrates a portion of a computer, including a CPU and conventional memory in which the invention may be embodied;

FIG. 2 illustrates a typical compiler showing the position of the code optimizer;

FIG. 3 illustrates a large scale organization of a code optimizer;

FIG. 4 illustrates a Data Dependency Graph constructed from a sample SBBN loop;

FIG. 5*a* illustrates the general characteristics and organization of computer instructions representing a non-optimized SBBN loop statement;

FIG. 5*b* illustrates the organization of computer instructions representing an SBBN loop optimized in accord with the invention having a compile-time determined tripcount;

FIG. 5*c* illustrates the organization of computer instructions representing an SBBN loop optimized in accord with the invention having an execute-time determined tripcount;

FIG. 6 Illustrates the compiler process for detecting an omega-invariant statement in a SBBN loop;

FIG. 7 Illustrates the general compiler process for optimizing execution of an omega- invariant statement within a SBBN loop using the invention;

FIG. 7*a* Illustrates the compiler process for creating an optimized SBBN loop having a compile-time determined tripcount; and

FIG. 7*b* illustrates the compiler process for creating an optimized SBBN loop with an execute-time determined tripcount.

NOTATIONS AND NOMENCLATURE

The detailed descriptions that follow are presented largely in terms of procedures and symbolic representation of operations on data bits within a computer memory. These procedural descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art.

A Data Dependency Graph (DDG) is a data structure in the computer memory that represents how statements within a loop depend on other statements. These graphs include nodes that represent computer operations and arcs that represent dependencies between the nodes. These dependencies include flow dependencies, data dependencies and anti-dependencies. Data structures within the compiler that represent data dependency graphs are used to determine loop-invariant and omega-invariant statements within a loop. These data structures are often represented by diagrams using circles for nodes corresponding to statements and arcs between the nodes representing dependencies. FIG. 4 illustrates an example DDG and is described below.

Instructions are the compiled binary operation codes (opcodes) for a target computer architecture that implement the operation described by a statement. Often one compiled statement will describe multiple operations and generate many computer instructions.

A loop operation, when compiled and the resulting computer instructions executed on a computer, causes the computer to repeatedly execute the instructions enclosed within the loop. Each repetition of the enclosed instructions is a single iteration of the loop.

An iterative construct is a series of operations that effectuate a loop operation defined by a loop statement and the statements it encloses.

A loop is a programming language construct that describes an iterative process where statements within the body of the loop define operations that are repeatedly executed by a computer. In other words a compiled loop, when executed in a computer, causes the computer to repetitively iterate through the operations described by statements contained in the loop until some completion condition is satisfied. As such, loop statements represent an iterative construct that provides an iterative control process coupled to other statements contained within the body of the loop. Loops, as optimized by the invention, are limited to single-basic-block-natural-loops (SBBN loops) that have a determinable number of iterations (i.e., having a compile-time computable or known symbolic tripcount). That is, loops that do not contain any control flow structures, functions, procedures, or other constructs that change the flow of execution within the loop. Such loops have only one entry point, one exit point, and no branches within the loop.

The compiled instructions that implement loops are organized into a prologue that initializes the loop and a kernel that contains the opcodes representing the compiled statements within the loop's body and the test for the completion condition on the loop. The flow of execution is such that the prologue instructions are executed before the kernel instructions. Thus loop-invariant operations within the loop's kernel can be moved (hoisted) to the loop's prologue with the same functional result as if they were left within the loop. Hoisting is accomplished by using standard code motion algorithms such as the one described in Section 10.7 of Aho.

A loop is unwound when the operations within the loop's kernel, for some number of initial iterations, are hoisted to the prologue. Thus, some loop iterations are peeled off the loop's kernel. The number of iterations peeled off of the loop is termed the peelfactor. For each peel, the kernel operations are duplicated. For loops that have a variable number of iterations (e.g., that the iterations are determined at execution-time and not at compile-time), two versions of the instructions that effectuate the loop are created. One version effectuating a loop where the omega-invariant operation remains

within the kernel and the other version including peeled iterations, an optimized omega-invariant operation and the optimized kernel. If the number of iterations specified for a particular invocation of the loop is less then the number of iterations peeled from the loop, the original loop is selected. Otherwise, the optimized version of the loop, consisting of the peels, the hoisted omega-invariant operation and the kernel, is executed.

The tripcount is the number of iterations required of the loop at the time the loop is entered. If the tripcount is greater than the peelfactor, the modified iterative process (the peels, the hoisted omega-invariant operation and the optimized kernel) will be selected, otherwise the original unpeeled loop will execute. For loops that have a tripcount that can be determined at compile time, the compiler determines if and how much to peel the loop.

A loop-invariant statement is a statement within a loop that does not depend on any other variant statement in the loop and generates the same result in each iteration of the loop. Prior art techniques exist to hoist compiled instructions, corresponding to loop-invariant statements, to the loop's prologue.

An omega-invariant statement is a statement that is initially variant for some omega iterations of a loop, but that stabilizes and remains invariant after omega iterations of the loop.

An operation is described by a statement and is represented by the corresponding intermediate code. The back-end portion of a compiler converts the operations described by the intermediate code into sequences of executable instructions for a target computer architecture. These instructions, when executed on the target computer effectuate the operation.

A procedure is a self-consistent sequence of steps leading to a desired result. These steps are those requiring physical manipulation of physical quantities. Usually these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. These signals are referred to as bits, values, elements, symbols characters, terms, numbers, or the like. It will be understood by those skilled in the art that all of these and similar terms are associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

Statements are the textual strings that have the syntax and semantics of a programming language. An example statement is "while (TRUE) {} ;" an instruction implementing the operation described by this statement would be the binary opcode for the target computer that causes the computer to repeatedly execute the same instruction. In some assembler language, this operation could be represented by "A: BR A".

The manipulations performed by a computer in executing opcodes are often referred to in terms, such as adding or comparing, that are commonly associated with mental operations performed by a human operator. No such capability of a human operator is necessary in any of the operations described herein that form part of the present invention; the operations are machine operations. Useful machines for performing the operations of the invention include programmed general purpose digital computers or similar devices. In all cases the method of computation is distinguished from the method of operation in operating a computer. The present invention relates to method steps for operating a computer in processing electrical or other (e.g., mechanical, chemical) physical signals to generate other desired physical signals.

The invention also relates to apparatus for performing these operations. This apparatus may be specially constructed for the required purposes or it may comprise a general purpose computer as selectively activated or reconfigured by a computer program stored in the memory of a computer. The procedures presented herein are not inherently related to a particular computer or other apparatus. In particular, various general purpose machines may be used with programs written in accordance with the teachings herein, or it may prove more convenient to construct more specialized apparatus to perform the required method steps. The required structure for a variety of these machines will appear from the description below.

Finally, the invention may be embodied in a computer readable medium encoded with an optimizing compiler application program.

DESCRIPTION OF THE PREFERRED EMBODIMENT

Operating Environment

The invention can be practiced with any programming language that utilizes loop constructs. Some of these programming languages include, but are not limited to, FORTRAN, PASCAL, C, C++, ADA and compiled BASIC. Example loop constructs in C and C++ are the "for", the "do-while", and the "while" statements. The optimization provided by the invention applies to single-basic-block-natural-loops having a determinable number of iterations. Thus, these loops have either a tripcount determined at compile-time or a tlipcount determined at execute-time from a known operation. These loops have only one entry point, one exit point, and no branching of the flow of execution within the loop.

The invention is used with a computer system. Some of the elements of such a computer system are shown in FIG. 1, wherein a processor 1 is shown, having an Input/Output ("I/O") section 2, a central processing unit ("CPU") 3 and a memory section 4. The I/O section 2 is connected to a keyboard 5, a display unit 6, a disk storage unit 9 and a CD-ROM drive unit 7. The CD-ROM unit 7 can read a CD-ROM medium 8 that typically contains programs 10 and data. Such a computer system is capable of executing a compiler program that embodies the invention.

FIG. 2 illustrates a typical optimizing compiler 20, comprising a front-end compiler 24, a code optimizer 26 and a back-end code generator 28. The front-end 24 of a compiler takes, as input, a program written in a source language 22 containing a series of statements, including loop statements, and performs various lexical, syntactical and semantic analysis on these statements creating an intermediate code 32 representing the operation of the target program. This intermediate code 32 is used as input to the code optimizer 26 that attempts to adjust (improve) the intermediate code resulting in a faster-executing set of machine instructions 30. Some code optimizers 26 are trivial while others do a variety of computations in an attempt to produce the most efficient target program possible. Those of the latter type are called "optimizing compilers" and include such code transformations as common sub-expression elimination, dead-code elimination, hoisting loop-invariant statements from a loop's kernel, renaming temporary variables and interchanging two independent adjacent statements as well as performing register allocation. The intermediate code is either adjusted or recreated during the optimization process. Finally, the adjusted intermediate code is passed to the back-end code generator 28 where the operations described by this intermediate code are converted to binary opcodes corresponding to specific instructions for the target computer architecture.

FIG. 3 depicts a typical organization of an optimizing compiler 40. On entry of the intermediate code 42, a data dependency graph is constructed 44. At this stage the code transformations mentioned above take place 46. Next, if supported by the compiler, instruction scheduling or "pipelining" may take place 48, followed by "Register allocation" 50. Finally, the adjusted intermediate code is sent 52 to the compiler back-end for conversion to the binary instructions of the target machine architecture.

The invention can be practiced in other segments of a compiler. For example, the invention can be practiced during the semantic and lexical scanning phase of the compiler when generating the initial intermediate code representing the operations of the program. This is true so long as the invention has access to sufficient information to construct a data dependency graph for a program loop; can determine either the tripcount, or a variable containing the tripcount, of the program loop; and can determine the operations enclosed within the program loop. In particular, in one embodiment, the invention resides in the instruction scheduling or software pipelining module 48. However, another embodiment could effectively have the invention reside within other various optimization portions 46 of the compiler. In fact, the invention could be practiced in compilers that do not use the "front-end, optimizer, back-end" structure.

The present invention increases the execution speed of target programs by reducing the number of instructions repeatedly executed during some programmed loops. It identifies program loop statements that contain omega-invariant statements and optimizes the execution speed of these loops. For example, the following SBBN loop (a "C" language "for" statement having a compile-time tripcount):

```
x = 1;
for (i = 0; i < =10; i++) {
  y = x * 5;
  a[i] = x;
  b[i] = y;
  x = 2;
}
```

would be optimized by the invention as if written as:

```
x = 1;
  y = x * 5;                    //peel 1 (iteration 0)
  a[0] = x;
  b[0] = y;
  x = 2;
  y = x * 5;                    //peel 2 (iteration 1)
  a[1] = x;
  b[1] = y;
for (i = 2; i < = 10; i++) {    // iteration 2–9
  a[i] = x;
  b[i] = y;
  }
```

further, the following loop (containing an execute-time tripcount):

```
x = 1;
for (i = 0; i < = N; i++) {
  y = x * 5;
  a[i] = x;
  b[i] = y;
  x = 2;
}
```

would be optimized by the invention as if written as:

```
x = 1;
if (N > 2) {
    x = 1;
      y = x * 5;                     //peel 1 (iteration 0)
      a[0] = x;
      b[0] = y;
      x = 2;
      y = x * 5;                     //peel 2 (iteration 1)
      a[1] = x;
      b[1] = y;
    for (i = 2; i < = N; i++) {      // iteration 2-N
      a[i] = x;
      b[i] = y;
      } else {                       // if N < =2
      for (i = 0; i < = N; i++) {
      y = x * 5;
      a[i] = x;
      b[i] = y;
      x = 2;
      }
}
```

Thus, in one aspect of the invention, single-basic-block-natural loops having a determinable number of iterations and containing omega-invariant operations are optimized during compilation, by unwinding omega iterations from the loop and then hoisting the stabilized omega-invariant operations from the kernel of the peeled loop. To determine which statements are omega-invariant, the compiler must first construct a data dependency graph.

FIG. 4 illustrates a data dependency graph (DDG) 400 for a loop described as shown by the source code at label 401. As is well known in the art, this source is representative of a SBBN loop asdescribed using the "C" language "for" statement. Such a loop statement represents an iterative construct when complied into computer instructions. DDGs interrelate how statements in a loop depend on other statements. Nodes 402, 404, 406 in the graph represent the three statements enclosed in the loop, arcs 408, 410 represent the data dependencies between the nodes. Here both node A 402 and node C 406 depend on node B 404. The numbers 412 next to the arcs 408, 410 indicate the data dependency distance between the nodes. Thus, arc 408 between node B 404 and node A 402 has a data dependency distance of 1 indicating that the assignment to x in one iteration is used by node A in the next iteration. Compare this with the arc 410 between node B 404 and node C 406 that has a dependency distance of zero. Arc 410 indicates that the assignment to x in node B 404 is used by node C 406 in the same iteration. Thus, a data dependency graph represents which operations depend on other operations. The data dependency distance is used when determining an omega for an operation.

FIGS. 5*a*, 5*b*, and 5*c* illustrate different ways a loop operation can be implemented. FIG. 5*a* illustrates the organization of the instructions that effectuate a loop that has not been optimized by the invention. FIG. 5*b* illustrates the organization for loops having a compile-time determined number of iterations and optimized by the invention. FIG. 5*c*

illustrates the organization for loops having an execute-time determined number of iterations and optimized by the invention. These figures are further described below.

FIG. 5a illustrates the general organization of the iterative construct formed by the instructions that effectuate the loop operation described by a loop statement. When executing in a computer and as indicated by the arrow labeled 523, the loop operation starts when the flow of execution enters the prologue 501 of the iterative construct. The prologue 501 contains instructions that initialize loop control variables and effectuate loop-invariant operations. For a SBBN loop, the instructions in the prologue are executed on entry into the loop. The iterative construct includes a kernel 503 that contains instructions that are repeatedly executed in sequence every iteration of the loop until the loop termination condition occurs. If the termination condition has not occurred, the kernel 503 repeats for another iteration. Thus, the computer again executes the instructions in the kernel 503 as indicated by the arrow labeled 505. When the loop operation terminates, the flow of execution continues as indicated by the arrow labeled 507.

FIG. 5b illustrates the general organization of the instructions, as optimized by the invention, that effectuate the loop operation described by a SBBN loop statement having a compile-time determined number of iterations (the tripcount). The optimized SBBN loop retains the same basic characteristics as shown in FIG. 5a. The prologue 507 of the optimized loop contains the same operations as did the prologue 501 of the loop in FIG. 5a. However, the prologue 507 is expanded by the invention and now includes a number of peeled iterations 509 and a region that contains hoisted omega-invariant operations 511. Thus, the prologue 507 again contains instructions that initialize loop control variables and effectuate loop-invariant operations. After these instructions 507 execute, the flow of execution moves to the peels 509 (the iterations) unwound by the invention. Each of these peels 509 consist of the same operations as in the kernel 503 including any loop bookkeeping operations. In particular, peel #1 510 contains the instructions to effectuate the operations for the first iteration of the SBBN loop, peel #2 512 contains the instructions to effectuate operations for the second iteration and so forth through peel #n 514.

When executed, peels #1 through #n 509 effectuate the first omega iterations for the iterative construct's operation. That is, these peels include omega-invariant operations that have not stabilized. The number of peels corresponds to the peelfactor determined during the optimization process. After executing the peels 509, the flow of execution moves to the hoist region 511 where instructions that effectuate stabilized omega-invaliant operations are executed. The optimized kernel 513 no longer includes the omega-invariant operation as this operation has been hoisted to the hoist region 511. Because the optimized kernel 513 no longer includes the omega-invaliant instructions the optimized kernel 513 executes faster then the kernel 503 described in FIG. 5a. When the loop operation terminates, the flow of execution continues as indicated by the arrow labeled 533.

FIG. 5c illustrates the general organization of the instructions, as optimized by the invention, that effectuate the loop operation described by a SBBN loop statement having an execute-time determined number of iterations (the tripcount). Because the tripcount is determined at execution time, the iterative construct comprising the loop operation contains two versions of the loop operation—one version contains an optimized kernel 513 and the other contains a non-opfimized kernel 503. The compiler inserts additional operations 517 to select which version to use. These operations are dependent upon the tripcount required for any particular invocation of the loop operation. When this optimized loop operation executes, the flow of execution enters the prologue 507 as indicated by the arrow marked 523, having the same characteristics as the prologue 507 shown in FIG. 5b. After initializing required loop control variables and executing loop-invariant instructions, the flow of execution moves to the tripcount test 517. The instructions in the tripcount test 517 determine, by comparing the tripcount to the peelfactor, whether the optimized version of the SBBN loop operation is executed as indicated by the arrow marked 521. If this path 521 is taken, execution occurs through the peels 509, the hoist region 511, and the optimized kernel 513 in the same manner as was described for FIG. 5b. The flow of execution leaves the SBBN loop operation as indicated by the arrow labeled 525.

If the tripcount test 517, determines that non-optimized loop operation is to execute, the flow of execution follows the path indicated by the arrow marked 519 and the loop operation occurs in the same fashion as was described for the loop shown in FIG. 5a. Those skilled in the art will recognize that many techniques exist to place the instructions to select which loop operation to use and that these instructions can be placed either before or after the prologue.

FIGS. 6 and 7 illustrate how the invention is practiced in a compiler to generate the optimized loop operations having the structure shown in FIGS. 5b and 5c.

FIG. 6, illustrates the general process 600 of an optimizing compiler implementing an omega-invariant statement detection mechanism configured to detect omega-invariant operations in a SBBN loop. To perform this operation, the compiler detects omega-invariant operations within the loop operation. Once these operations are detected, an omega-invariant statement determination mechanism determines the omega for the statement. The flow of execution begins 602 when this segment of the optimizing compiler receives incoming intermediate code representing a SBBN loop. When the compiler detects a SBBN loop 603 it constructs a data dependency graph from the intermediate code that represents that loop 604 and initializes the peelfactor for the loop to zero. Each node on the graph is initialized as variant with a stabilizer of zero 606. Then, using the DDG, all loop-invariant operations are marked invariant 608 so that they will be hoisted out of the loop's kernel and into the iterative construct's prologue. This is accomplished using prior art techniques such as those described in Aho.

Next, each variant node is examined 610. If the examined node has only one arc, from a node in the loop marked invariant—the source node, the examined node is also marked invariant 612. The omega (the stabilizer value of the operation) for the examined node is set to the maximum of the existing omega for the examined node and the omega for the source node plus the dependency distance on the arc from the examined node to the source node. Finally, the peelfactor is set to the maximum of its current value and the omega for the current node.

Now at step 614, if any node was marked invariant during the just completed pass over the variant nodes, the process repeats by again examining each variant node 610. If no new nodes were marked invariant during the just completed pass, the detection process completes through 616.

FIGS. 7, 7a and 7b illustrate how the invention optimizes the omega-invariant operations detected as shown in FIG. 6.

FIG. 7 illustrates the general process 700 of an optimizing compiler implementing the invention for hoisting omega-invariant operations from the loop's kernel. This process, when performed by a computer, comprises a loop optimi-

zation mechanism. This mechanism adjusts the intermediate code representation of the loop operation, based on the omega values determined by the techniques described with FIG. **6**, so as to optimize the loop operation. The compiler first determines whether the number of iterations for the loop is known at compile time. If so, then the optimizer creates an iterative construct of the form shown in FIG. **5***b*. If the number of iterations for the loop will only be known during execute-time, then the optimizer operates to create in iterative construct of the form shown in FIG. **5***c*. Hoisting of omega-invariant operations begins **702** upon receiving a DDG and the intermediate code that has been marked as explained in the description of FIG. **6**. Next, the compiler determines whether the tripcount of the loop can be determined during compilation or is to be determined during execution **704**. If the tripcount can be determined at compilation time the compiler processes **710** the loop as illustrated by FIG. **7***a*. If the tripcount can only be determined at execution time, the compiler processes **712** the loop as illustrated by FIG. **7***b*. In either case, the hoisting mechanism completes through the exit block **714** producing adjusted intermediate code that reflects the modifications made by the invention. The operations described by this adjusted intermediate code are eventually converted to binary opcodes specific to a target computer architecture with the structure as indicated in FIG. **5***b* or **5***c*.

FIG. **7***a* illustrates the mechanism used by the compiler that optimizes loop operations by adjusting the intermediate code, when the loop's tripcount can be determined at compile time. The hoisting process begins **710** upon receiving a DDG and the intermediate code. Then **714**, loop invariant operations are hoisted to the prologue. Next **716**, the compiler, using the loop peeling mechanism, unwinds peelfactor iterations (peelfactor is the maximum omega for all the statements within the loop) from the kernel, leaving omega peeled iterations and a kernel having a stabilized omega-invariant operation. Then **718**, the compiler hoists the stabilized omega-invariant operations to the hoist region from the kernel, using prior art methods and techniques, leaving an optimized kernel. Finally, this process exits through the block labeled **719**. One skilled in the art will recognize that it is within the scope of the invention to hoist operations that stabilize before peelfactor so as to optimize subsequent peeled iterations.

FIG. **7***b* illustrates how the compiler hoists operations when the loop's tripcount can not be determined at compile time. In this circumstance, the tripcount is determined during the flow of execution. When the iterative construct representing the loop operation is entered, the tripcount is evaluated, and either a non-optimized loop operation selected for further execution (if the tripcount is less than the number of peels) or, if the tripcount is large enough, an optimized loop operation is selected.

The compiler's hoisting process begins **722** upon receiving a DDG and the intermediate code. Then **724**, loop invariant operations are hoisted to the prologue. Because the tripcount is determined at execution time, the iterative construct effectuates two versions of the loop operation -one version contains the prologue, peels, the hoisted omega-invariant operations and the optimized kernel (the optimized loop operation) and the other contains only the prologue and the non-optimized kernel. The compiler contains a loop selection insertion mechanism that inserts **726** additional operations to select which version of the loop operation to use. These selection operations are dependent upon the tripcount required by any given invocation of the loop operation. These selection operations are inserted in the prologue and comprise a comparison of the tripcount with the peelfactor. If, at execution, the tripcount is greater then the peelfactor, the peels, the hoisted omega-invariant operations and the kernel are executed. Otherwise, the loop operation containing the unpeeled kernel is executed. Those skilled in the art will recognize that the scope of the invention encompasses an embodiment that creates additional optimized loop operations using differing peelfactors along with the additional program logic to select between them.

Once the selection logic is inserted in the adjusted intermediate code representation, the SBBN loop operation, as defined by the intermediate code, is duplicated **728** by a loop duplication mechanism. One of the duplicates is then processed **730** by a loop peeling mechanism as explained in the description of FIG. **7***a*, above, to peel peelfactor iterations from the kernel, leaving omega peeled iterations of the loop's kernel and the kernel having a stabilized omega-invariant operation. Finally **732**, the stabilized omega-invariant operations are hoisted, by a hoisting mechanism, to the hoist region from the kernel leaving an optimized kernel and the process exits **734**. This process leaves an iterative construct in the form shown in FIG. **5***c* and explained above. Again, one skilled in the art will recognize that it is within the scope of the invention to hoist operations that stabilize before peelfactor so as to optimize subsequent peeled iterations.

One skilled in the art will understand that the invention as described above teaches a computer system for optimizing programmed loop statements, an apparatus for optimizing programmed loop statements, a code optimizer for use with a compiler, a computer controlled method, and a computer program product embodied in a computer readable program code mechanism.

Further, one skilled in the art will understand that various modifications and alterations may be made in the preferred embodiment disclosed herein without departing from the scope of the invention. Accordingly, the scope of the invention is not to be limited to the particular invention embodiments discussed above, but should be defined only by the claims set forth below and equivalents thereof.

What is claimed is:

1. A computer system having a central processing unit (CPU) and a random access memory (RAM) coupled to said CPU, for use in compiling a target program for use with a target computer architecture, said computer system comprising:

(a) a compiler system resident in said computer system having an optimization mechanism comprising:

(b) a loop-invariant statement optimization mechanism configured to optimize a loop statement containing a loop-invariant statement, said loop statement representing an iterative construct and said loop statement having the characteristics of a single basic block natural loop and including a determinable number of iterations;

(c) an omega-invariant statement detection mechanism configured to detect a statement that is variant for at least the first iteration of said loop statement but which becomes invariant before the completion of the execution of all iterations of said loop statement within said loop statement;

(d) said omega-invariant statement determination mechanism being further configured to determine an omega such that after omega iterations said omega-invariant statement becomes invariant, said omega-invariant statement determination mechanism including

i. a loop detection mechanism configured to duplicate said iterative construct leaving said iterative construct and a duplicated iterative construct having a kernel;

ii. a loop peeling mechanism configured to unwind omega iterations from said kernel leaving one or more peeled iterative and said kernel having a stabilized omega-invariant operation;

iii. a hoisting mechanism configured to hoist said stabilized omega-invarialit operation out of said kernel; said peeled iterations and said hoisted omega-invariant operation and said kernel comprising an optimized iterative construct; and

(e) an omega-invariant statement optimization mechanism configured to optimize said iterative construct responsive to said omega.

2. The computer system of claim 1 wherein said determinable number of iterations is determined at execute-time, said loop optimization mechanism further comprising:

a loop selection insertion mechanism configured to insert an execute-time loop selection mechanism, said loop selection mechanism configured to select either said optimized iterative construct or said duplicated iterative construct responsive to said execute-time determined number of iterations.

3. An apparatus for optimizing execution time for executable instructions in a target program that is designated to run on a target computer architecture, said apparatus comprising:

(a) a computer having a processor, a memory and an input/output section;

(b) a compiler system resident in said computer memory having an optimization mechanism comprising:

(c) a loop-invariant statement optimization mechanism configured to optimize a loop statement containing a loop-invariant statement, said loop statement representing an iterative construct and said loop statement having the characteristics of a single basic block natural loop and including a determinable number of iterations;

(d) an omega-invariant statement detection mechanism configured to detect a statement that is variant for at least the first iteration of said loop statement but which becomes invariant before the completion of the execution of all iterations of said loop statement, said omega-invariant statement determination mechanism including

i. a loop detection mechanism configured to duplicate said iterative construct leaving said iterative construct and a duplicated iterative construct having a kernel;

ii. a loop peeling mechanism configured to unwind omega iterations from said kernel leaving one or more peeled iterative and said kernel having a stabilized omega-invariant operation;

iii. a hoisting mechanism configured to hoist said stabilized omega-invariant operation out of said kernel; said peeled iterations and said hoisted omega-invariant operation and said kernel comprising an optimized iterative construct; and

(e) said omega-invariant statement determination mechanism being further configured to determine an omega such that after omega iterations said omega-invariant statement becomes invariant; and

(f) an omega-invariant statement optimization mechanism configured to optimize said iterative construct responsive to said omega.

4. The apparatus of claim 3 wherein said determinable number of iterations is determined at execute-time, said omega-invariant statement optimization mechanism further comprising:

a loop selection insertion mechanism configured to insert an execute-time loop selection mechanism, said loop selection mechanism configured to select either said optimized iterative construct or said duplicated iterative construct responsive to said execute-time determined number of iterations.

5. A code optimizer for use in a compiler system for compiling a target program to run on a target computer architecture, said code optimizer comprising:

(a) a first portion configured to accept, as input, an intermediate code representation representing an iterative construct, said iterative construct having the characteristics of a single basic block natural loop including a determinable number of iterations;

(b) a second portion, coupled to said first portion, configured to optimize a loop statement containing a loop-invariant statement;

(c) a third portion, coupled to said first portion, configured to detect an omega-invariant operation within said iterative construct said omega-invariant operation being variant for at least the first iteration of said iterative construct but which becomes invariant before the completion of said iterative construct and to determine an omega such that after omega iterations said omega-invariant operation becomes invariant, said third portion including

i. a loop detection mechanism configured to duplicate said iterative construct leaving said iterative construct and a duplicated iterative construct having a a kernel;

ii. a loop peeling mechanism configured to unwind omega iterations from said kernel leaving one or more peeled iterative and said kernel having a stabilized omega-invariant operation;

iii. a hoisting mechanism configured to hoist said stabilized omega-invariant operation out of said kernel; said peeled iterations and said hoisted omega-invariant operation and said kernel comprising an optimized iterative construct;

(d) a fourth portion, coupled to said third portion and configured to optimize said iterative construct responsive to said omega; said second and fourth portions generating an adjusted intermediate code representation; and

(e) a fifth portion configured to output said adjusted intermediate code representation.

6. The code optimizer of claim 5 wherein said determinable number of iterations is determined at execute-time, and wherein said fourth portion further comprises:

a sixth portion to insert an execute-time loop selection mechanism to select either said optimized iterative construct or said duplicated iterative construct; said loop selection mechanism responsive to said determinable number of iterations, said sixth portion, said seventh portion, said eighth portion, and said ninth portion generating said adjusted intermediate code representation.

7. A computer controlled method of optimizing a loop statement within a target program directed at a target computer architecture, said loop statement describing an iterative construct, said loop statement having the characteristics of a single basic block natural loop including a detenninable number of iterations, said method comprising the steps of:

(a) detecting said loop statement containing one or more body statements;

(b) optimizing said loop statement with respect to a loop-invariant statement if such exists within said loop statement;

(c) detecting that one of said body statements is an omega-invariant statement that is variant for at least the first iteration of said loop statement but which becomes invariant before the completion of the execution of all iterations of said loop statement;

(d) determining an omega associated with said omega-invariant statement such that after omega iterations said omega-invariant statement becomes invariant, said step of determining including the sub-steps of:

(i) duplicating said iterative construct to provide a said iterative construct and a duplicated iterative construct having a kernel;

(ii) unwinding omega iterations from said kernel to provide one or more peeled iterative and said kernel having a stabilized omega-invariant operation;

(iii) hoisting said stabilized omega-invariant operation out of said kernel such that said peeled iterations and said hoisted omega-invariant operation and said kernel comprising an optimized iterative construct; and

(e) optimizing said iterative construct responsive to said omega.

8. The method of claim 7 wherein said determinable number of iterations is determined at execute-time, and wherein step (e) further comprises:

inserting an execute-time loop selection mechanism configured to select either said optimized iterative construct or said duplicated iterative construct, said loop selection mechanism responsive to said execute-time determined number of iterations.

9. A computer program product comprising:

(a) a computer usable medium having a computer readable program code mechanism embodied therein to optimize a target program containing executable instructions directed toward a target computer architecture, said computer readable program code mechanisms in said computer program product comprising:

(b) computer readable code first optimization mechanisms to cause a computer to optimize a loop statement containing a loop-invariant statement, said loop statement representing an iterative construct and said loop statement having the characteristics of a single basic block natural loop and including a determinable number of iterations;

(c) computer readable code detection mechanisms to cause said computer to detect an omega-invariant statement that is variant for at least the first iteration of said loop statement but which becomes invariant before the completion of the execution of all iterations of said loop statement within said loop statement, said code detection mechanisms being configured to perform the operations of:

(i) duplicating said iterative construct to provide a said iterative construct and a duplicated iterative construct having a kernel;

(ii) unwinding omega iterations from said kernel to provide one or more peeled iterative and said kernel having a stabilized omega-invariant operation;

(iii) hoisting said stabilized omega-invariant operation out of said kernel such that said peeled iterations and said hoisted omega-invariant operation and said kernel comprising an optimized iterative construct;

(d) computer readable code determination mechanisms to cause said computer to determine an omega associated with said omega-invariant statement such that after omega iterations said omega-invariant statement becomes invariant;

(e) computer readable code omega-invariant statement optimization mechanisms to cause said computer to optimize said iterative construct responsive to said omega.

10. The computer program product of claim 9 wherein said determinable number of iterations is determined at execute-time in which said computer readable code omega-invariant statement optimization mechanisms comprise:

computer readable code insertion mechanisms to cause said computer to insert an execute-time loop selection mechanism, said loop selection mechanism configured to select either said optimized iterative construct or said duplicated iterative construct responsive to said execute-time determine number of iterations.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 6,026,240

DATED : February 15, 2000

INVENTOR(S) : Krishna Subramanian

It is certified that error appears in the above-identified patent and that said Letters Patent are hereby corrected as shown below:

Column 16, line 33, change "having a a kernel" to --having a kernel--.

Column 16, line 66, change "detenninable" to --determinable--.

Signed and Sealed this

Fifteenth Day of May, 2001

Attest:

Nicholas P. Godici

NICHOLAS P. GODICI

Attesting Officer

Acting Director of the United States Patent and Trademark Office

US006490721B1

(12) **United States Patent**
Gorshkov et al.

(10) **Patent No.: US 6,490,721 B1**
(45) **Date of Patent: Dec. 3, 2002**

(54) **SOFTWARE DEBUGGING METHOD AND APPARATUS**

(75) Inventors: **Vassili Gorshkov**, Centreville, VA (US); **Richard Efron**, Chevy Chase, MD (US); **Andrew Jolyon Platt**, Fairfax, VA (US); **Paul William Kohlbrenner**, Fairfax, VA (US)

(73) Assignee: **OC Systems Incorporated**, Fairfax, VA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/350,177**

(22) Filed: **Jul. 9, 1999**

Related U.S. Application Data

(60) Provisional application No. 60/092,796, filed on Jul. 14, 1998.

(51) **Int. Cl.**[7] **G06F 9/45**

(52) **U.S. Cl.** **717/130**; 717/129

(58) **Field of Search** 717/130, 129, 717/176, 178; 703/23

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,802,165 A		1/1989	Ream	714/38
4,819,233 A		4/1989	Delucia et al.	717/129
5,142,679 A		8/1992	Owaki et al.	717/151
5,193,180 A		3/1993	Hastings	717/163
5,265,254 A		11/1993	Blasciak et al.	717/130
5,274,811 A		12/1993	Borg et al.	717/128
5,307,498 A		4/1994	Eisen et al.	717/154
5,313,616 A		5/1994	Cline et al.	717/127
5,335,344 A		8/1994	Hastings	714/35
5,408,650 A		4/1995	Arsenault	717/124
5,465,258 A		11/1995	Adams	717/130
5,499,340 A		3/1996	Barritz	714/47
5,528,753 A		6/1996	Fortin	714/35
5,535,329 A		7/1996	Hastings	714/35
5,539,907 A		7/1996	Srivastava et al.	717/130
5,581,696 A		12/1996	Kolawa et al.	714/38
5,581,697 A		12/1996	Gramlich et al.	714/35
5,583,988 A		12/1996	Crank et al.	714/48
5,590,056 A		12/1996	Barritz	702/186
5,619,678 A		4/1997	Yamamoto	711/165
5,619,698 A		4/1997	Lillich et al.	717/168
5,623,665 A		4/1997	Shimada et al.	714/5
5,659,752 A		8/1997	Heisch et al.	717/158
5,675,803 A		10/1997	Preisler et al.	717/131
5,675,804 A		10/1997	Sidik et al.	717/139
5,689,684 A	*	11/1997	Mulchandani et al.	703/23
5,694,566 A		12/1997	Nagae	711/1
5,710,724 A		1/1998	Burrows	714/34
5,732,273 A		3/1998	Srivastava et al.	717/128
5,815,653 A	*	9/1998	You et al.	717/134
5,860,012 A	*	1/1999	Luu	709/220
6,044,224 A	*	3/2000	Radia et al.	709/331
6,292,934 B1	*	9/2001	Davidson et al.	717/158
6,353,923 B1	*	3/2002	Bogle et al.	717/128

OTHER PUBLICATIONS

Rational Purify; Internet website; www.rational.com; 5 pages (Oct. 8, 1999).

Bug Trapper; Internet website; www.bugtrapper.com; 4 pages (Oct. 8, 1999).

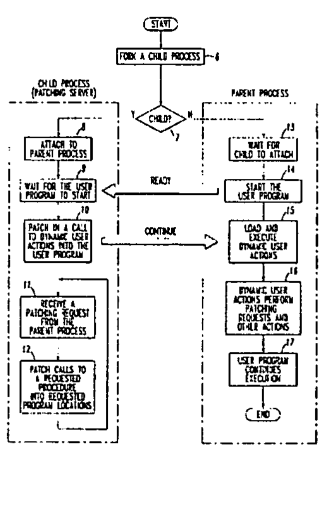
* cited by examiner

Primary Examiner—Gregory Morse
Assistant Examiner—John Q. Chavis
(74) *Attorney, Agent, or Firm*—Nixon Peabody LLP; Donald R. Studebaker

(57) **ABSTRACT**

A method and apparatus for debugging software for the purpose of modification of the target program's behavior and/or collection of data pertinent to a target program's execution. New user actions are compiled and converted into a dynamically linkable module. The existing program is run under the control of a dynamic action linker. The dynamic action linker modifies the existing program by inserting the new actions in the memory image. The insertion is accomplished by automatically recognizing and modifying object code sequences in the existing program to call the new actions. Once the modification phase has finished modifying the existing program's memory image the new program is run without additional interruption, the new actions acting as if they were present in the original source code for the program.

13 Claims, 4 Drawing Sheets



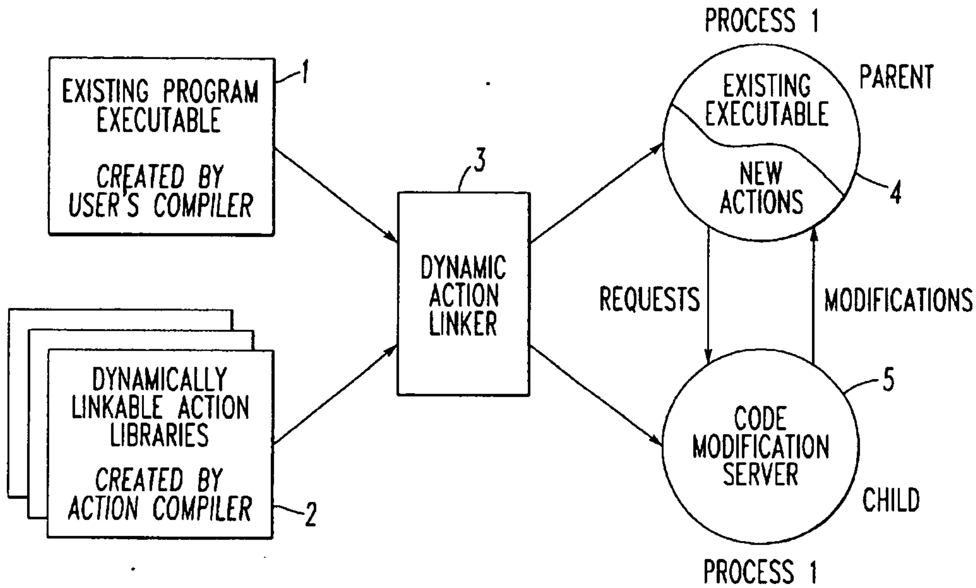

PROCESS 1
EXISTING PROGRAM EXECUTABLE
CREATED BY USER'S COMPILER
1
EXISTING EXECUTABLE
PARENT
NEW ACTIONS
3
4
DYNAMIC ACTION LINKER
REQUESTS
MODIFICATIONS
DYNAMICALLY LINKABLE ACTION LIBRARIES
CREATED BY ACTION COMPILER
2
5
CODE MODIFICATION SERVER
CHILD
PROCESS 1

FIG. 1

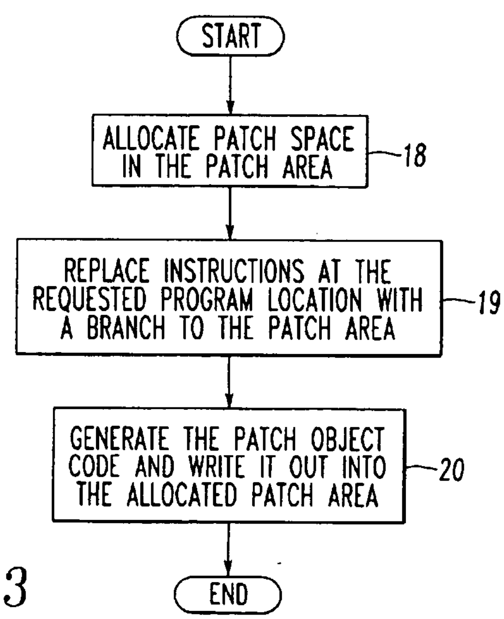

START
ALLOCATE PATCH SPACE IN THE PATCH AREA
18
REPLACE INSTRUCTIONS AT THE REQUESTED PROGRAM LOCATION WITH A BRANCH TO THE PATCH AREA
19
GENERATE THE PATCH OBJECT CODE AND WRITE IT OUT INTO THE ALLOCATED PATCH AREA
20
END

FIG. 3

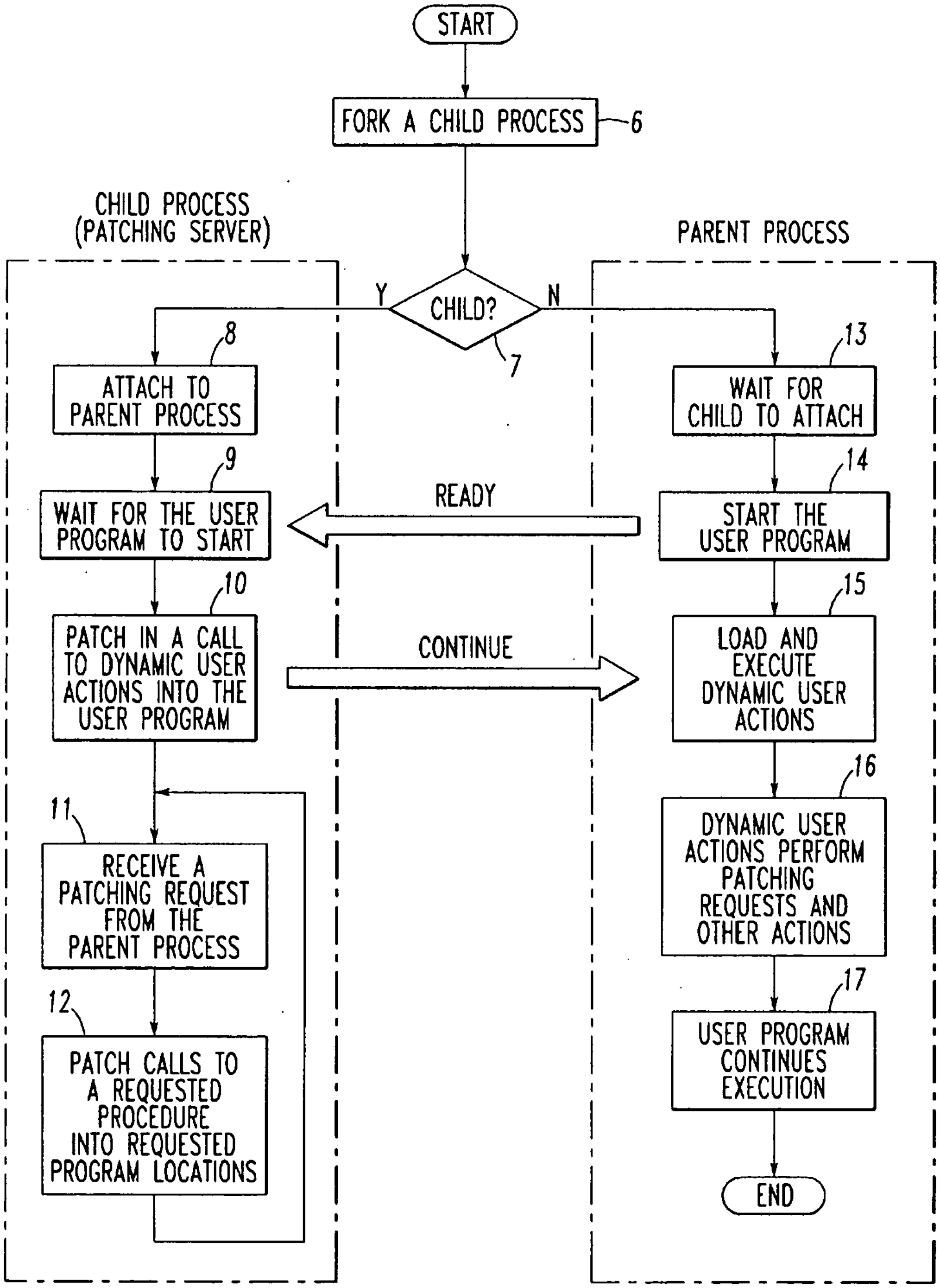
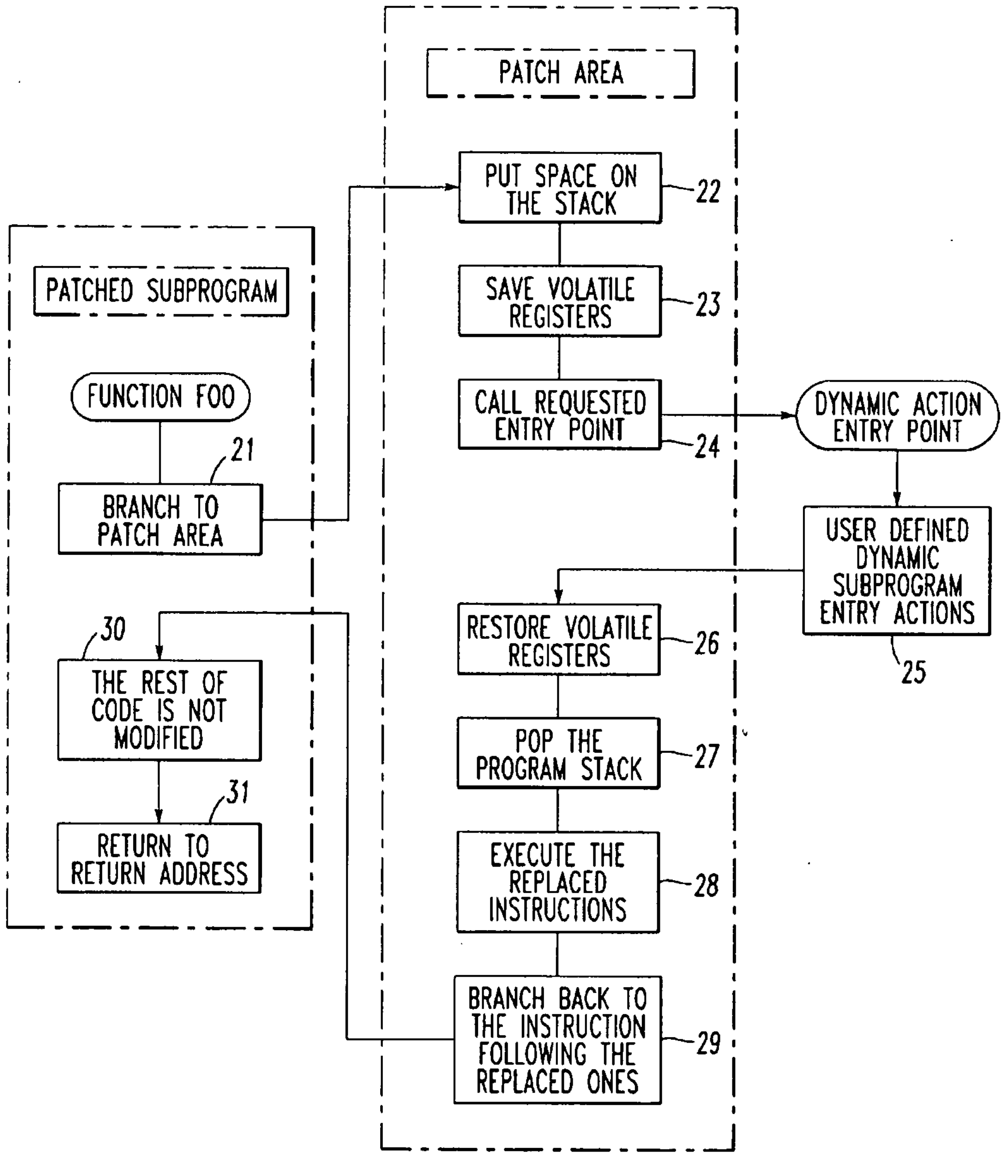

START
FORK A CHILD PROCESS
6
CHILD PROCESS (PATCHING SERVER)
PARENT PROCESS
CHILD?
7
Y
N
8
ATTACH TO PARENT PROCESS
9
WAIT FOR THE USER PROGRAM TO START
10
PATCH IN A CALL TO DYNAMIC USER ACTIONS INTO THE USER PROGRAM
11
RECEIVE A PATCHING REQUEST FROM THE PARENT PROCESS
12
PATCH CALLS TO A REQUESTED PROCEDURE INTO REQUESTED PROGRAM LOCATIONS
READY
CONTINUE
13
WAIT FOR CHILD TO ATTACH
14
START THE USER PROGRAM
15
LOAD AND EXECUTE DYNAMIC USER ACTIONS
16
DYNAMIC USER ACTIONS PERFORM PATCHING REQUESTS AND OTHER ACTIONS
17
USER PROGRAM CONTINUES EXECUTION
END

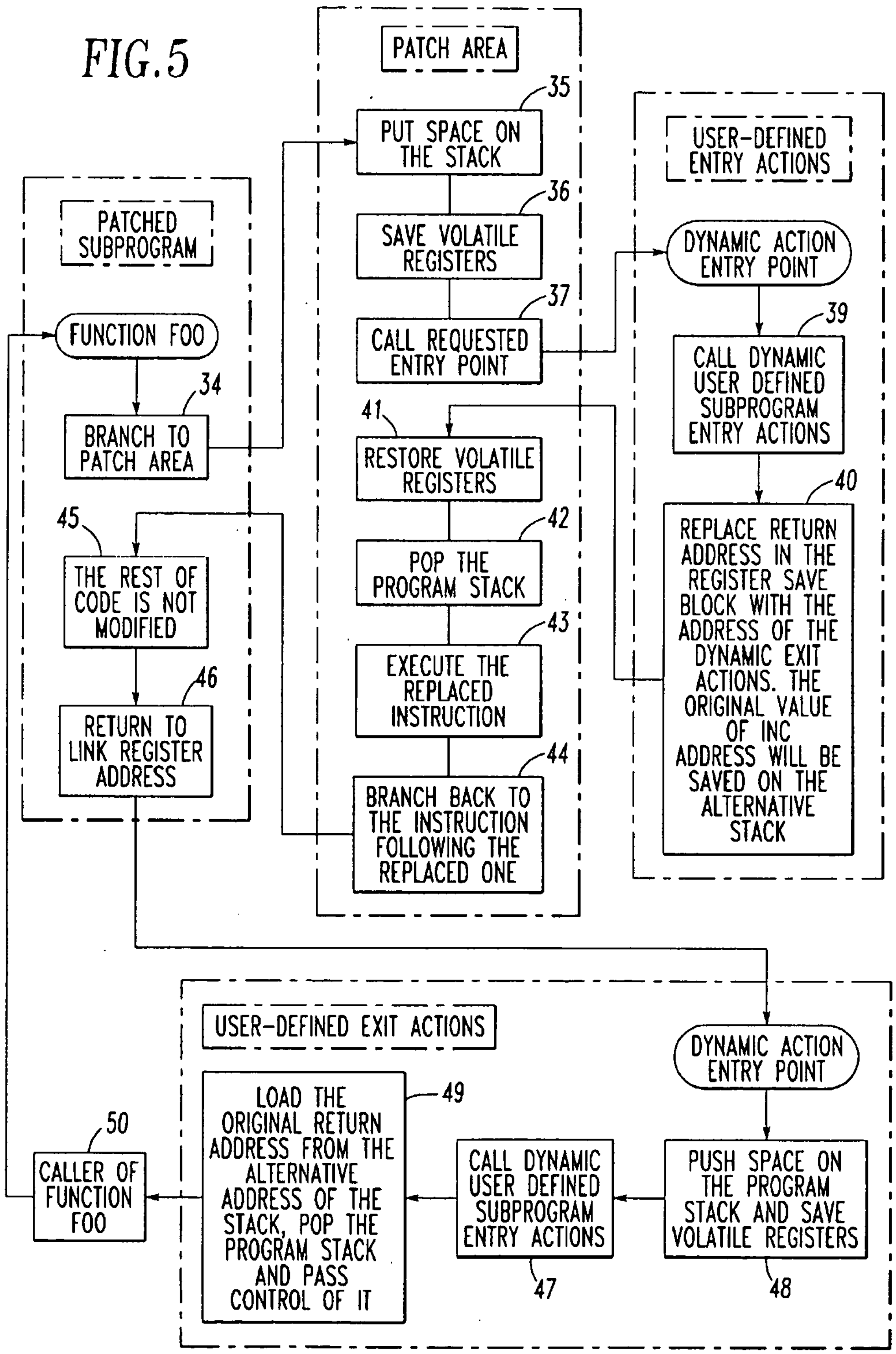
FIG.2



PATCH AREA
PUT SPACE ON THE STACK
22
SAVE VOLATILE REGISTERS
23
CALL REQUESTED ENTRY POINT
24
DYNAMIC ACTION ENTRY POINT
USER DEFINED DYNAMIC SUBPROGRAM ENTRY ACTIONS
25
PATCHED SUBPROGRAM
FUNCTION FOO
21
BRANCH TO PATCH AREA
RESTORE VOLATILE REGISTERS
26
30
THE REST OF CODE IS NOT MODIFIED
POP THE PROGRAM STACK
27
31
RETURN TO RETURN ADDRESS
EXECUTE THE REPLACED INSTRUCTIONS
28
BRANCH BACK TO THE INSTRUCTION FOLLOWING THE REPLACED ONES
29

FIG. 4

FIG.5



PATCH AREA
35
PUT SPACE ON THE STACK
36
SAVE VOLATILE REGISTERS
37
CALL REQUESTED ENTRY POINT
41
RESTORE VOLATILE REGISTERS
42
POP THE PROGRAM STACK
43
EXECUTE THE REPLACED INSTRUCTION
44
BRANCH BACK TO THE INSTRUCTION FOLLOWING THE REPLACED ONE
PATCHED SUBPROGRAM
FUNCTION FOO
34
BRANCH TO PATCH AREA
45
THE REST OF CODE IS NOT MODIFIED
46
RETURN TO LINK REGISTER ADDRESS
USER-DEFINED ENTRY ACTIONS
DYNAMIC ACTION ENTRY POINT
39
CALL DYNAMIC USER DEFINED SUBPROGRAM ENTRY ACTIONS
40
REPLACE RETURN ADDRESS IN THE REGISTER SAVE BLOCK WITH THE ADDRESS OF THE DYNAMIC EXIT ACTIONS. THE ORIGINAL VALUE OF INC ADDRESS WILL BE SAVED ON THE ALTERNATIVE STACK
USER-DEFINED EXIT ACTIONS
DYNAMIC ACTION ENTRY POINT
48
PUSH SPACE ON THE PROGRAM STACK AND SAVE VOLATILE REGISTERS
47
CALL DYNAMIC USER DEFINED SUBPROGRAM ENTRY ACTIONS
49
LOAD THE ORIGINAL RETURN ADDRESS FROM THE ALTERNATIVE ADDRESS OF THE STACK, POP THE PROGRAM STACK AND PASS CONTROL OF IT
50
CALLER OF FUNCTION FOO

SOFTWARE DEBUGGING METHOD AND APPARATUS

RELATED APPLICATION DATA

This application claims benefit of Provisional Patent Application Serial No. 60/092,796, filed on Jul. 14, 1998, the disclosure of which is incorporated herein by reference.

FIELD OF THE INVENTION

The invention relates to debugging of computer code. More specifically, the invention relates to a method and apparatus for debugging computer code that does not require modification of the target executable program that is to be debugged.

DESCRIPTION OF THE RELATED ART

Computer programming has become a very complex operation because of the increased sophistication of computers and the increased complexity of tasks to be carried out by computers. Further, computer hardware is now relied upon for mission critical tasks, such as air traffic control, communications, medical equipment, and the like. It follows that computer software, i.e. a computer program, is controlling these tasks. Note that the phrase "computer program" is used herein to refer to a set of software instructions and/or routines for controlling computer hardware. The word "application" is used herein to refer to the practical use of the program, for example, inventory control, accounting or any other use.

Of course, a higher level of software complexity results in a higher potential for programming errors, known as "bugs". For example, many programs consist of thousands or even millions of lines of code. The sheer volume of code creates great potential for bugs. Also, as a result of the size of most computer programs, the code often is developed by plural teams of programmers each working on a different portion or aspect of the program. The teams may accomplish similar tasks in a different manner or otherwise produce code that is incompatible with code developed by other teams. Due to the huge reliance on computer software in just about every aspect of society, it is important to find and eliminate as many bugs as possible in an efficient manner. Also, it is often desirable to insure the performance of a computer program. For example, the end user may want to be guaranteed that a particular variable is updated very quickly.

To find errors in code and assure performance, a process called "debugging" has developed. One form of conventional debugging is called "interactive debugging". Interactive debugging includes stopping the program at desired break points, known as "breaks", for examination of program status. "Noninteractive debugging" does not stop the program. However it uses debugging statements which are inserted into the source code and the source code is then compiled into machine code including the debugging statements. Once all errors have been eliminated and the program is operating in a desirable manner, the debugging statements are removed from the source code and the program is recompiled and used for its intended task.

Conventional debugging techniques, while sometimes effective, have several shortcomings. In particular, conventional noninteractive debugging techniques require access to the source code for insertion of the debugging statements. Often, the user license for a particular program does not grant access to the source code or the right to change the source code. Therefore, in most cases, the user must rely on

the developer of the software to provide all debugging. Often programming errors occur only in a particular application of a program and thus are not caught by the developer during initial debugging. Also, the developer may be unavailable or unwilling to provide subsequent debugging services. Further, even if access to the source code is not an obstacle, the program as run during noninteractive debugging has been altered to accommodate the debugging statements. Therefore, the program might not run in the same manner during debugging as it will during normal operation. The debugging statements may mask errors, create errors, slow operation, or otherwise affect the function or performance of the program. Finally, many programs behave differently or even refuse to run when stopped. Therefore, conventional interactive debugging using break points is not reliable and is difficult to implement.

SUMMARY OF THE INVENTION

It is an object of the invention to overcome the limitations of conventional debugging techniques.

It is another object of the invention to permit debugging of a computer program without stopping execution of the program.

It is another object of the invention to permit noninteractive debugging of a computer program without accessing or altering the source code of the program.

It is another object of the invention to conduct performance testing of a computer program under actual executable conditions of the program.

It is another object of the invention to conduct requirement verification under actual executable conditions of the program.

It is another object of the invention to increase the speed and efficiency of software development.

It is another object of the invention to facilitate software comprehension.

The invention uses "dynamic linking" to use the existing target program executable for debugging and thus avoids the long edit-compile-link cycles inherent in the use of debugging statements in source code. By linking new user actions into the executable image, the target program is allowed to run as it would normally run thus allowing debugging of time sensitive programs without the need to stop execution thereof. Calls to the user actions are inserted into the memory image during loading of the target program. The calls can be inserted at any time during or after transfer of the executable code into memory. Accordingly, the term "loading" as used herein refers to any time during or after transfer of the target program into memory for the purpose of running the program. The invention can be used to locate and fix programming errors, for requirements verification, for performance evaluation, for software comprehension, or for any other evaluation of software. Accordingly, the term "debugging" as used herein refers broadly to any type of diagnosis or evaluation of software.

A first aspect of the invention is a method for debugging a computer program comprising the steps of developing a debugging subprogram having a user action for debugging a target program, loading the target program for execution, inserting a call to the debugging subprogram into a memory image of the target program during the loading step, and executing the target program.

A second aspect of the invention is a computer readable medium having instructions for debugging a computer program recorded thereon. The medium a includes a first set of

instructions for loading the target program for execution, and a second set of instructions for inserting a call to a debugging subprogram having user actions into a memory image of the target program during loading of the target program.

BRIEF DESCRIPTION OF THE DRAWING

The invention is described through a preferred embodiment and the attached drawing in which:

FIG. 1 is a schematic diagram illustrating the primary components of a preferred embodiment of the invention;

FIG. 2 is a flow chart illustrating the operation of the dynamic action linker in accordance with the preferred embodiment;

FIG. 3 is a flow chart illustrating the steps performed to patch a call into the target program;

FIG. 4 is a flow chart illustrating the steps performed to patch code into the parent process by the child process in accordance with the preferred embodiment; and

FIG. 5 is a flow chart illustrating the steps performed to patch code into the parent process by the child process when user actions are requested for entry and exit of a routine in accordance with the preferred embodiment.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

FIG. 1 illustrates the primary components of the preferred embodiment. Target program **10** is the existing program in executable format that is to be debugged. Target program **10** is a machine code executable file that has been compiled from source code in a known manner using a known software compiler. User action libraries **12** are created by compiling one or more debugging subprograms **16** (containing user actions) into linkable libraries (similar to Dynamic Link Libraries (DLL) in a Windows™ environment). This compiling procedure is accomplished by action compiler **14** which compiles the source code of subprograms **16**, which are written in ANSI C for example, into machine code of user action libraries **12**. These files together with target program **10** are input to dynamic action linker **18**. Note that there can be multiple user action libraries **12**, including a runtime library.

Dynamic action linker **18** reads target program **10** and user action libraries **12** and creates two processes. A first process **20** created by dynamic action linker **18** consists of target program **10** and the debugging user actions needed from user action libraries **12**. A second process **22** created by dynamic action linker **18** handles requests from process **20**, to modify code locations in process **20** as described in detail below. Action compiler **14** and dynamic action linker **18** can be recorded as instructions on a computer readable medium.

FIG. 2 illustrates the operation of dynamic action linker **18**. In step A, dynamic action linker **18** forks a child process to create a parent process which corresponds to process **20** disclosed above and a child process which corresponds to process **22** above. The fork system code is a well known service supported by most operating systems in which a copy known as a "child" is made of parent program in RAM. Child process **22** allows attachments of debugging commands. In step B, both parent process **20** and child process **22** identify themselves and the execution path splits. Child process **22** then attempts to attach to parent process **20** using known available operating system services in step C. Child process **22** waits for an acknowledgment from parent process **20** that it has successfully attached in step D. In step E, child process **22** has received the acknowledgment from parent process **20** of a successful attachment and continues on to patch into parent process **40** a call to the dynamic user actions runtime start routines, i.e. the debugging actions. In particular, calls to user actions in user action libraries **12** are inserted in the memory image of target program **10** (which is running as will be seen below) and user action libraries **12** are loaded into RAM in a separate area. In step F, child process **22** waits for a new request for patching service from parent process **20**. In step G, child process **22** has received a request from parent process **20** and executed the request. Steps F and G are repeated until the task is terminated by parent process **20**.

Of course, parent process **20** runs simultaneously with steps C–G of child process **22**. In step H, parent process **20** waits until child process **22** has attached to parent process **20** using the operating system debug facility. In step I, parent process **20** loads the user program, i.e. replaces the current memory image with target program **10** in a known manner. Since child process **22** has control of parent process **20**, the operating system loads target program **10** and holds parent process **20** while it notifies child process **22**. Child process **22** patches, in step E, in such a way that it will load and branch dynamic user actions to target program **10** in step J. The patched target program **10** then continues to run in step K with the new dynamic user actions from user action libraries **12**. After completing the patched user actions, target program **10** continues executing in step L.

FIG. 3 illustrates Step E of FIG. 2, i.e., patching of dynamic user actions into target program **10** in detail. Child process **22** created by dynamic action linker **18** must patch the memory image of target program **10** so that it will call the newly loaded user action routines. In step E1, child process **22** allocates space for the patch in the patch area in parent process **20**. In step E2, child process **22** replaces an instruction (or instructions) at the requested program location with a branch instruction to the patch area. In step E3, child process **22** generates code to call the user action.

FIG. 4 illustrates the function of the code patched into the parent process **20** by child process **22**. In step M, a subroutine of target program **10** begins. In step AB, a subroutine call placed in the subroutine branches to patch area **42**. In patch area **42**, code to do the following actions has been written: allocate space on the patch stack (AC), save any registers to the patch stack (AD), call the dynamic user action routine (AE) enter the dynamic user action (AF), execute the dynamic user action and return to the patch area (AG), restore any saved registers from the patch stack (AH), return the space allocated from the patch stack (AI), execute the instruction or instructions that were removed from the subroutine entry sequence (AJ), and return to subroutine (AK). In step AL remaining code is executed and execution is returned to the caller of the subroutine in step AM.

FIG. 5 illustrates the function of parent process **20** and the child process **22** when user actions have been requested for the exit as well as the entry to a routine. In step BA, a subroutine begins. In step BB, a call has been inserted in the subroutine and thus it branches into patch area **42**. In patch area **42**, code to do the following actions has been written: allocate spaced on the patch stack (BC), save any registers to the patch stack (BD), enter the user action (BF), call the user action routine (BE), execute the user action and return to the patch area (BG), replace the subroutine's return location with dynamic exit action patch routine (BH), restore any saved registers from the patch stack (BI), return the space allocated from the patch stack (BJ), execute the instruction or instructions that were removed from the

subroutine entry sequence (BK), and return to subroutine (BL). In step BM, the rest of the subroutine is executed. In step BN execution returns to the dynamic exit action routing (BN). A user action also begins in step (BO). Space on the patch stack is allocated and any registers are saved to the patch stack in step BP. The dynamic user action is executed in step BQ. The saved registers are restored and the space allocated on the patch stack is returned in step BR. In step BQ execution returns to the caller of the subroutine in step BJ.

The various components and functions described above can be in the form of computer software running on a computer such as a standard personal computer or a server. The software can be written with known programming languages and stored on known media. For example, magnetic media, such as a removable diskette or a hard drive can be used. Also, the software can be stored on optical media, such as CDROM. Alternatively, any computer readable medium can be used to store the software of the preferred embodiment. The invention has been described through a preferred embodiment. However, various modifications can be made without departing from the scope of the invention and defined by the appended claims.

The invention allows the exact executable that will be shipped to a customer to be tested/verified. The invention can be used for subprogram call tracing, logging subprogram calls and parameter values, performance measurements, memory usage tracking and detections of memory violations, fault injection, test coverage, component testing, program and data verification, and requirements verification.

What is claimed is:

1. A method for debugging a computer program comprising the steps of:

developing a debugging subprogram having a user action for debugging a target program;

loading the target program for execution;

inserting a call to the debugging subprogram into a memory image of the target program during said loading step; and

executing the target program with the call inserted therein.

2. A method as recited in claim 1, wherein said developing step comprises the steps of:

preparing source code of a debugging routine: and

compiling the source code of the debugging routine into the debugging subprogram.

3. A method as recited in claim 2, wherein said inserting step comprises the steps of:

allocating patch space in the debugging subprogram;

replacing instructions of the debugging subprogram with a branch to the patch space; and

generating object code to call the user action.

4. A method as recited in claim 1, wherein said step of executing the target program comprises the steps of:

executing the debugging subprogram when the call to the debugging subprogram is encountered in the target program; and

returning to execution of the target program after said step of executing the debugging subprogram.

5. A method as recited in claim 1, wherein said step of executing the target program comprises the steps of:

forking a child process of the debugging subprogram; and

loading the target program with the parent process of the debugging subprogram.

6. A method as recited in claim 5, wherein said step of inserting comprises patching a call to a dynamic user action into the target program with the child process.

7. A method as recited in claim 2, wherein said preparing step comprises inserting a user action into the source code to reference entities of the target program.

8. A method as recited in claim 7, wherein said entities comprise at least one of variables, functions, expressions and objects declared in the target program.

9. A computer readable medium having instructions for debugging a computer program recorded thereon, said medium comprising:

a first set of instructions for inserting a call to a debugging subprogram containing a user action into a memory image of the target program during loading of the target program; and

a second set of instructions for executing the target program with the call inserted therein.

10. A medium as recited in claim 9, wherein said first set of instructions comprises:

instructions for allocating patch space in the debugging subprogram;

instructions for replacing instructions of the debugging subprogram with a branch to the patch space; and

instructions for generating patch object code to call the user action.

11. A medium as recited in claim 9, wherein said second set of instructions comprises:

instructions for executing the debugging subprogram when the call to the debugging subprogram is encountered in the target program; and

instructions for returning to execution of the target program after executing the debugging subprogram.

12. A medium as recited in claim 9, wherein said second set of instructions further comprises:

instructions for forking child process of the debugging subprogram; and

instructions for loading the target program with the parent process of the debugging subprogram.

13. A medium as recited in claim 12, wherein said first set of instructions further comprises instructions for patching a call to a dynamic user action into the target program with the child process.

* * * * *

US005193190A

United States Patent [19]

Janczyn et al.

[11] **Patent Number: 5,193,190**

[45] **Date f Patent: Mar. 9, 1993**

[54] **PARTITIONING OPTIMIZATIONS IN AN OPTIMIZING COMPILER**

[75] Inventors: **Joyce M. Janczyn**, Toronto, Canada; **Peter W. Markstein**, Austin, Tex.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **371,487**

[22] Filed: **Jun. 26, 1989**

[51] **Int. Cl.**[5] **G06F 9/45**

[52] **U.S. Cl.** **395/700**; 364/DIG. 1; 364/280.4; 364/280.5

[58] **Field of Search** .... 364/200, 900, DIG. 1, 364/DIG. 2; 395/700

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,435,753	3/1984	Rizzi	364/200
4,506,325	3/1985	Bennett et al.	364/200
4,567,574	1/1986	Saadé et al.	364/900
4,571,678	2/1986	Chaitin	364/200
4,642,764	2/1987	Auslander et al.	364/200
4,642,765	2/1987	Cocke et al.	364/200
4,656,582	4/1987	Chaitin et al.	364/200
4,656,583	4/1987	Auslander et al.	364/200
4,773,007	9/1988	Kanada et al.	364/200
4,782,444	11/1988	Munshi et al.	364/200
4,953,084	8/1990	Meloy et al.	364/200
4,961,141	10/1990	Hopkins et al.	364/200

OTHER PUBLICATIONS

Gries, D., *Compiler Construction for Digital Computers*, pp. 375–411 (1971).

J. T. Schwartz, "On Programming" An Interim Report on the SETL Language. Installment II: The SETL Language and Examples of its Use, Courant Institute of Math Sciences, NYU 1973, pp. 293–310.

E. Morel and C. Renvoise, "Global Optimization by Suppression of Partial Redundancies", CACM, vol. 22, No. 2, pp. 96–103, 1979.

A. Aho and J. Ullman, "Principles of Compiler Design", Addison Wesley, 1977, pp. 13–19, 406–477.

Proceedings of the Sigplan Symposium on Compiler Construction, vol. 14, No. 8, Aug. 6, 1979, Denver, US, pp. 214–220, J. E. Ball, "Predicting the Effects of Optimization on a Procedure Body (Program Compilers)", the whole document.

Decus Proceedings of the Spring Symposium, May 12, 1969, Wakefield, US, pp. 103–104, D. R. Donati et al, "Techniques for Compiling Large Fortran Programs for PDP-9 Computer", p. 103, column 2, line 11–p. 104, column 1, line 12.

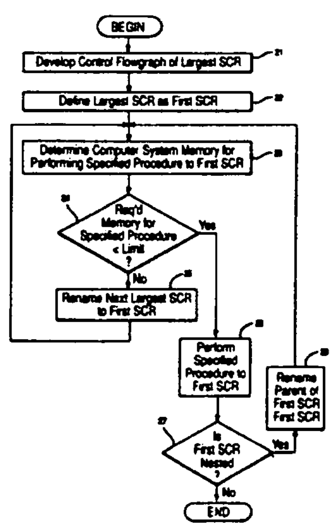
Hewlett-Packard Journal, vol. 37, No. 1, Jan. 1986, pp. 4–18, D. S. Coutant et al, "Compilers for the New Generation of Hewlett-Packard Computers", p. 6–p. 7, section Components of the Optimizer, p. 10, column 1, line 54 p. 10, column 2, line 13.

Primary Examiner—David L. Clark
Assistant Examiner—Matthew C. Fagan
Attorney, Agent, or Firm—Douglas H. Lefeve

[57] **ABSTRACT**

A computer program to be compiled is optimized prior to carrying out the final compilation. Subgraphs within the program are identified and examined for optimization beginning with the entire program as the largest subgraph. The number of entities in each subgraph which are relevant to each dimension of arrays used to represent data flow equations is determined. Next, the amount of memory required to contain the arrays is determined. If that memory requirement is within a predefined memory usage limit for the compilation, then a specified procedure of the compilation process is applied. If the memory requirement to contain the arrays exceeds the predefined memory usage limit for the compilation, the process is repeated for successively smaller subgraphs within the program in an attempt to find a subgraph to which the memory limits allow application of the specified procedure.

7 Claims, 2 Drawing Sheets



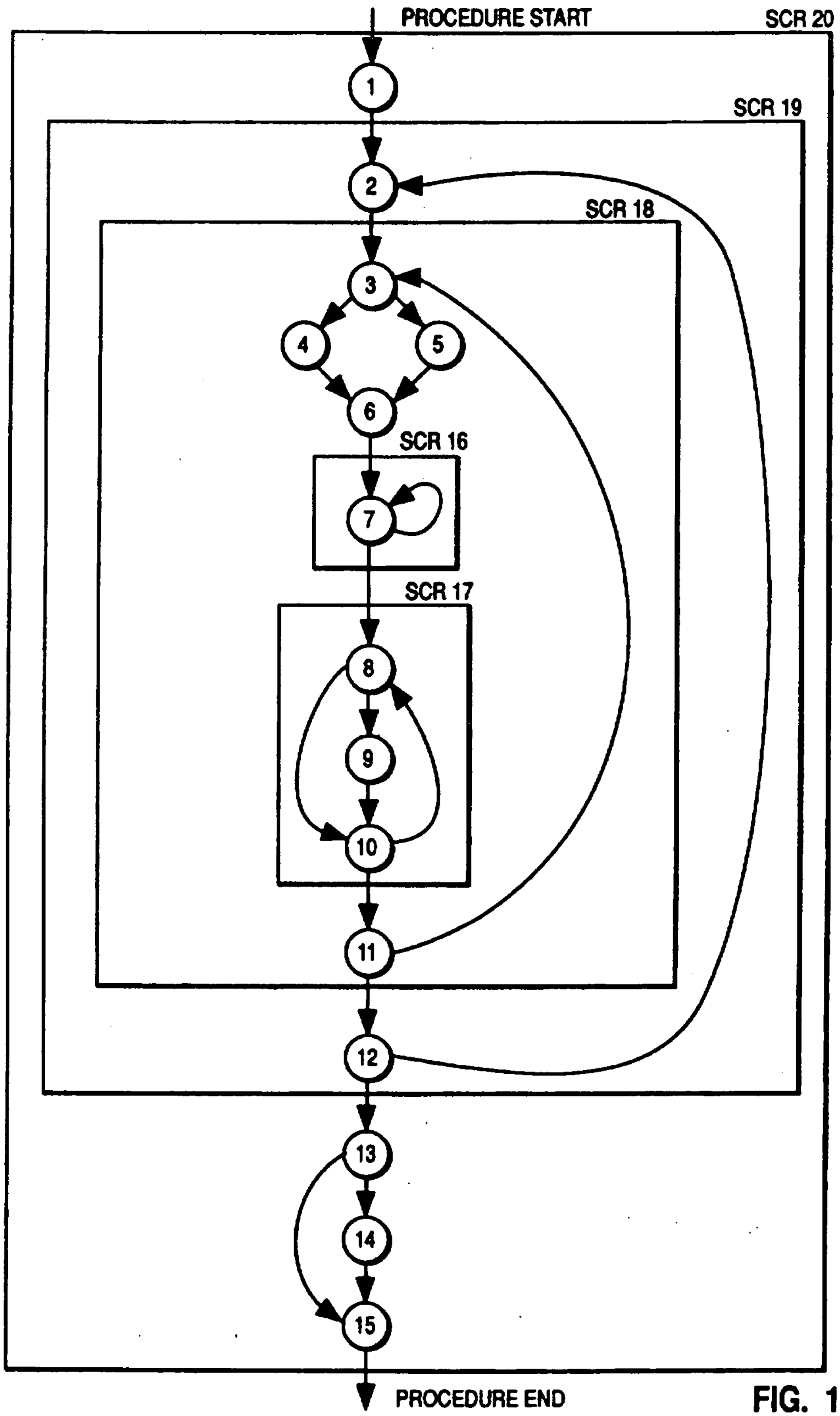

PROCEDURE START
SCR 20
SCR 19
SCR 18
SCR 16
SCR 17
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
PROCEDURE END

FIG. 1

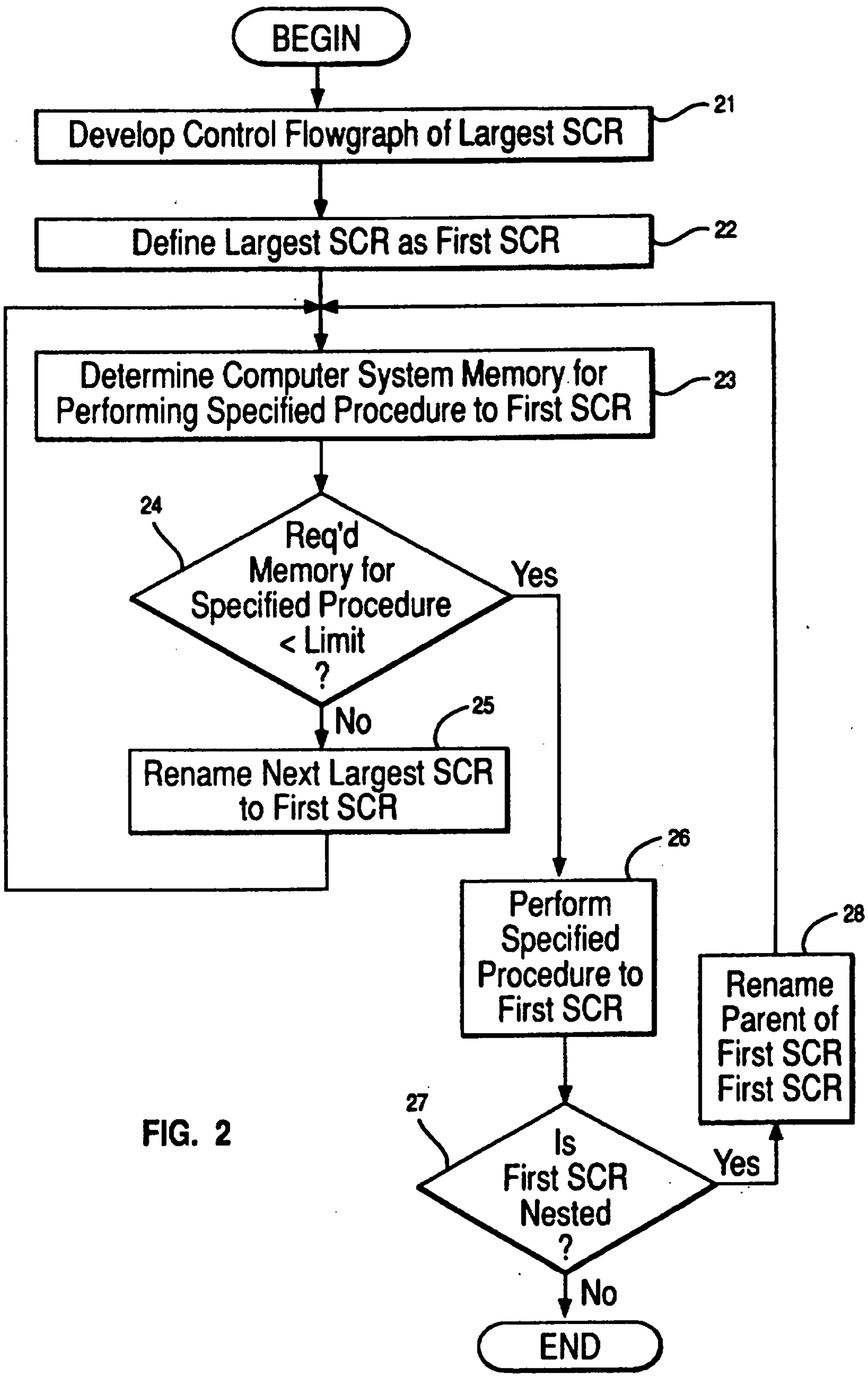

BEGIN
Develop Control Flowgraph of Largest SCR
21
Define Largest SCR as First SCR
22
Determine Computer System Memory for Performing Specified Procedure to First SCR
23
24
Req'd Memory for Specified Procedure < Limit ?
Yes
No
25
Rename Next Largest SCR to First SCR
26
Perform Specified Procedure to First SCR
28
Rename Parent of First SCR First SCR
27
Is First SCR Nested ?
Yes
No
END

FIG. 2

PARTITIONING OPTIMIZATIONS IN AN OPTIMIZING COMPILER

This invention relates to the improvement of the optimizing process in an optimizing complier. More particularly, it relates to a method for preventing the optimization process from being abandoned because of lack of memory space when compiling large programs in optimizing compilers that use global optimization algorithms (herein called "specified procedures") to improve the quality of the code that they generate.

BACKGROUND OF THE INVENTION

Optimizing compilers are important tools for programmers to improve the effectiveness and efficiency of the target CPU. The goal of an optimizing compiler is to generate the smallest and fastest set of object code possible that exactly duplicates the function of the program as it was written. In order to generate compact and efficient object code for computer programs written in high level languages, compilers used for such languages must utilize sophisticated global optimizers which generally use various specified procedures for reducing the run time or the memory requirements of a program. For example, a compiler may perform any or all of: common sub-expression elimination, code motion, strength reduction (replacing a slow operation by an equivalent fast operation), store motion and removing useless code sequences. Descriptions of some of these optimizations can be found in:

J. T. Schwartz, On Programming—An Interim Report on the SETL Language. Installment II: The SETL Language and Examples of its Use, Courant Institute of Math Sciences, NYU, 1973, pp. 293-310.

E. Morel and C. Renvoise, Global Optimization by Suppression of Partial Redundancies, CACM, Vol. 22, No. 2, pp. 96-103, 1979.

A. Aho and J. Ullman, Principles of Compiler Design, Addison-Wesley, 1977.

Each of these optimizing specified procedures transforms an intermediate language (IL) program into a semantically equivalent but more efficient IL program. Intermediate level language, as its name implies, is between a high level source program and machine code in complexity and sophistication. An intermediate level language can be especially useful in preparing compilers that are to be capable of translating any of several high level languages into machine code targetted to any of several machines; it reduces markedly the number of products that must be developed to cover a wide range of both machine types and programming languages, because all may translate through a common intermediate level language. It is at the intermediate language level that most optimizations are commonly performed.

The most important optimizations in an optimizing compiler are carried out globally, that is, on a program-wide level, rather than on a localized or basic block level. In performing each of these optimizations, a series of data flow equations must be solved. In doing so, the compiler gathers information about the expressions in the program being compiled; such information is dependent upon the flow of control in the program. For its own unique code transformation, each optimization must have a method of tracking when and how any given expression is available throughout the program as compiled.

This information is derived from the control flowgraph which is a directed graph depicting the possible execution paths of the program. In a low order flowgraph, the nodes represent basic blocks of a program and these are connected by directed edges representing paths along which control in the program flows. In a high order flowgraph the nodes are comprised of basic blocks and/or strongly connected regions.

In the present specification, the term "basic block" means any set of instructions in a computer program, whether object or source code, which is a straight-line sequence of code into which branches reach only its first instruction, and from which control leaves the basic block only after the last instruction.

The term "strongly connected region" means a set of nodes among which there is a path that can be repeatedly followed by the program control without passing through a node outside the region. Strongly connected regions are well known in the art of compiler design. A "single entry strongly connected region" is a strongly connected region that has only one node that is reached from outside the single entry strongly connected region. Hereafter in this disclosure, the term "region" means a single entry strongly connected region.

The term "subgraph" means any combination of nodes within the flowgraph. All strongly connected regions are also subgraphs, but not all subgraphs are strongly connected regions.

The term "entities" refers to the components of an intermediate representation which are used to describe a program as it is being compiled. These include variable entries, dictionary entries, results, expressions, instructions, and basic blocks of a program.

The larger and more complex a program is, the larger and more convoluted is its flowgraph, the greater the number of calculations involved and the greater the number of expressions for which dataflow equations must be solved. Memory requirements and processing time for the compilation tend to increase quadratically as a function of source program size for global optimization. When a situation arises where the compiler cannot optimize an entire program because of a space restriction, in the past the optimization has had to be abandoned. Attempts have been made to improve the quality of optimizations in the past. A small number of patents has been granted on inventions in this area.

U.S. Pat. No. 4,506,325 discloses a method of decreasing the storage requirements of a compiler by encoding operators and operands using an information theoretic encoding technique, applied to segments of a program. The disclosure does not deal with how a program is segmented.

U.S. Pat. No. 4,571,678 discloses a method of utilizing the limited number of registers in a target computer by improving register allocation procedures. It does not disclose any way of handling large programs that exceed the general memory availability of the target computer.

There remains a need for a program compilation technique that does not simply give up the struggle if an optimization cannot be performed within the constraints of the hardware or computer on which the program is being compiled.

It has now been discovered that the scope on which an optimization is applied can be limited, and yet many of the benefits of optimization can be realized. The program unit can be partitioned on the basis of its con-

trol flow structure into sections sufficiently small to be manipulated by the compiler.

BRIEF SUMMARY OF THE INVENTION

Accordingly, the present invention consists in a method, in an optimizing compiler using specified procedures for identifying ways of improving the quality of generated code, for optimizing a program to be compiled comprising, prior to carrying out an actual optimization:

(1) for a specified procedure, developing a control flowgraph representing all possible execution paths for said program;
(2) identifying subgraphs in said program;
(3) performing the steps of:
  (a) selecting a subgraph to be examined for optimization, said subgraph on a first iteration being the entire program;
  (b) by examining the code sequences in said subgraph, determining the number of entities in said subgraph which are relevant to each dimension of arrays used in said specified procedure to represent data flow equations;
  (c) determining the amount of memory required to contain said arrays;
  (d) if said amount of memory exceeds a predetermined memory usage limit for said compilation thereby denoting an unsuccessful attempt at optimizing the code in nodes of said subgraph, applying step (e) for said subgraph, otherwise carrying out said specified procedure on said subgraph; and
  (e) repeating steps (b) to (d) for every subgraph contained within said subgraph for which insufficient storage was found in step (d).

BRIEF DESCRIPTION OF THE DRAWING

FIG. 1 is a depiction of a typical flowgraph of a portion of a program, as would be built by a compiler according to the invention prior to each optimizing step that is to be performed.

FIG. 2 is a flow chart depicting the operation of the selection of appropriate subgraphs of a program for optimization according to this invention.

DESCRIPTION OF THE INVENTION

The first step in implementing the method of the invention is to construct a flowgraph of the program to be optimized. Flowgraph construction is well-known in the art and for the purposes of the invention, can be implemented by standard techniques. Then all of the appropriate subgraphs are identified within the flowgraph. The choice of subgraphs depends on the optimization procedure to be performed. Strongly connected regions are often useful, but intervals, hammocks and trees can also be used, and in extremely contained situations, a subgraph can consist of a single node. In the description of a preferred embodiment of the present invention the subgraphs are taken to be the collection of strongly connected regions.

The number and dimensions of arrays to be formed during the optimization must be known in order to write the specified procedure for the particular optimization. Their dimensions will be based upon the quantities of certain entities, for example instructions, results, expressions, dictionary entries, and basic blocks in the code sequence. The particular entities that are produced depend upon the specified procedure being undertaken. These quantities are constant for each subgraph of a program and the amount of memory needed can be easily calculated before the optimization is done. The memory limit that is imposed on the compiler can be pre-defined by the compiler writer at one or more levels or it can be dynamically determined as the compiler assesses the system's resource usage. One reason to constrain the amount of memory available to a selected procedure is also to constrain the execution time of the selected procedure. The memory requirements having been determined for applying the specified procedure to the entire program, the memory requirement is compared with the memory available and if the available memory is sufficient, then the optimization is carried out. If the memory is insufficient, then an attempt is made to apply the specified procedure to each of the maximal individual subgraphs within the subgraph for which there was insufficient space. For each contained subgraph for which there is still insufficient memory available, its contained subgraphs are optimized; if these contained subgraphs cannot be optimized, then each of the contained subgraphs will be further broken down in turn into its contained subgraphs. This iteration continues until a subgraph is examined whose optimization memory requirements are sufficiently small that an optimization can be carried out within the allotted memory, and the optimization is then performed. The method of the invention can be used with one or more optimizations, and when multiple optimizations are performed, they may be performed in any sequence.

An example of a specific optimization is the optimization technique known as common sub-expression elimination and code motion. For this optimization, the subgraphs into which the flowgraph is broken down are the strongly connected regions. One of the sets of data to be computed is the set of expressions that are available at the exit of each basic block and region within the outermost region being optimized (the "avail" set). In this case, two types of entities must be counted to determine the size of the "avail" set: the number of different expressions, and the number of basic blocks and regions including the region being commoned.

The availability information is collected by going through the code in the basic blocks of the outermost region being optimized (i.e. the low order control flowgraph) and marking which expressions are computed and not killed by any subsequent redefinitions of their arguments. These data are then propagated outward through the outer level regions to establish which expressions are available at the exits of every region. After this step, the assumption is made that nothing is available upon entry to the outermost region and this new information is propagated back into the inner regions. The need to process data flow information into and out of the regions of the flowgraph comes from the nature of the loops which comprise the regions. Clearly, common sub-expression elimination benefits from partitioning for large and complicated programs. The processing time for propagating information into and out of the basic blocks and regions of an entire program can be very large, and further, the space requirements of the arrays can be large. By partitioning the program according to the method of the invention, not only the second dimension of the "avail" set, but also its first dimension is affected. In most cases, the universe representing the set of expressions which appear in an inner region is a subset of the universe for the set of expressions found in its outer regions. Thus, by optimizing on

the next level of regions in a high order flowgraph, the size of an array can be significantly reduced because it is highly probable that more than one dimension is affected.

DETAILED DESCRIPTION OF THE INVENTION

A preferred embodiment of the invention will now be described with reference to FIG. 1, in which a flowgraph representing a portion of a typical compiled program is shown. In the embodiment illustrated here, the type of subgraph used is the strongly connected region. Nodes 1 through 15 represent basic blocks of one or more lines of code having one entry and one exit point. The entire flowgraph shown in FIG. 1 is a strongly connected region 20; this is defined as being true, even though it does not fit the classical definition because no return path exists from within the flowgraph to node 1.

Searching for the next inner strongly connected region can be done by methods known in the art, and is not critical to the method of the invention. In the present example node 2 has a path returning from node 12, but no return path from any later node. The next inner strongly connected region in the flowgraph is region 19, including nodes 2 through 12. The next contained strongly connected region within region 19 is analyzed similarly. Thus a path can be traced including nodes 3 through 11 repeatedly, and that defines region 18. Similarly, the other two strongly connected regions in the flowgraph of FIG. 1 are region 17, consisting of nodes 8, 9 and 10, and region 16, consisting of node 7 which has its own internal return path. Nodes 13 to 15 do not constitute a strongly connected region because there is no return path within the group of nodes that would permit a repeated flow of control without passing to a node outside the group.

The method of the invention is carried out on the program by, first, building the flowgraph shown in FIG. 1. Then the entire program is tested for an optimization procedure. As the compiler works through the entire program, a count of the entities used by the optimizing procedure is kept, including the number of entities which determine the size of the arrays which represent the data flow equations. As the counts are incremented, the unit size of each dimension of each array is noted and the total number of elements in the array is calculated.

This is then compared with the allotted memory. If the requirements exceed the allotted memory, then the next smaller strongly connected region is found as outlined above, and examined in the same manner. When a region is found that is sufficiently small to enable a successful optimization to be done, the optimization is performed on that region. The control flowgraph may optionally be rebuilt prior to optimizing on other specified procedures.

In some cases, it may be desirable, after a subgraph susceptible of being optimized has been treated, to attempt to optimize a larger subgraph. Since application of the selected procedure has reduced the size of the code in the subgraph to which it was applied, it may now be possible to treat the containing subgraph as a unit. However, such reexamination increases compilation time, whereas the thrust of the invention is to limit the compilation time and space during optimization. If carried out, such a step is performed in the same manner as the previous examination of the code sequences, except that successively higher-level regions are examined in turn.

The flow chart of FIG. 2 depicts each of the cases described above. The optimization method begins at 21 wherein a flowgraph of the largest strongly connected region is developed. At 22 the largest strongly connected region within this flowgraph (beginning with the entire program as the first defined strongly connected region, as described above) is set to the first SCR. At 23 the amount of computer system memory for performing the specified procedure of optimization to the first SCR is determined. At 24 a test is made to determine if the required memory for performing the specified procedure of optimization to the first SCR is less than the predetermined allowable limit of memory for this procedure. If not, at 25 the next largest SCR is renamed to the first SCR and the operation loops back to 23 to determine if the memory required for performing the specified procedure to this next largest SCR is less than the amount of available memory for this procedure. When an SCR is found so that the amount of required memory for performing the specified procedure is less than that available, at 26 the specified procedure of optimization is performed on that SCR, which had been named the first SCR. After performance of the specified procedure a test is made at 27 to determine if the first SCR is nested within a parent SCR. In not, the optimization process, with respect to the performance of this specified procedure, ends. If so, at 28 the parent SCR is renamed to the first SCR and the operation loops back to 23 to determine if the memory required for performing the specified procedure on this parent SCR is available.

Another example of the best mode of implementing the invention is given by the pseudo-code fragment in Appendix I. The code is adapted from an actual program, which operates as follows: The optimizer gets as input the memory limits which will constrain the optimizations. It resets the maps which are to be created to describe the universe for the sets of entities to be used to solve the dataflow equations for the optimization specified procedure, which here are commoning and code motion. It maps and counts the basic blocks which are contained in the given region, and the number of dictionary entries and different symbolic register uses found in the given region. It computes the size of the arrays used by the specified procedure. It compares this total with the allotted memory, and if the memory requirement exceeds the allotment, then returns the number which corresponds to the region whose nodes were most recently mapped. Else, it finds the regions at the next contained level. If such a region is found then it repeats the steps that were performed previously on the contained region. If the region contains no regions itself, then the optimization searches for the next region at the same level. If there is no region at the same level, then it looks for regions at the next higher level. It repeats ascending through the flowgraph levels until a region is found or else there are no more levels to ascend. If there are no more levels, it reports to the specified procedure that no regions to be optimized are found in the program. Comments within the code also help to illustrate the method of the invention.

The invention may be used on any machine using an intermediate level language in the process of compilation, including Reduced Instruction Set Computers (known generically as RISC computers). It can also be used with a compiler that is in turn used with many

different computer systems, by adjusting the memory limits to be utilized by the optimizer. By the same means, the user can obtain varied levels of optimization of the same source program. Because the optimizing occurs at the intermediate language level, it can also be used with optimizing compilers dealing with any high level language, including Fortran, PL/I, Cobol, C, SETL and Pascal. Through the use of the invention, at least partial optimizations can be performed upon very large programs that would be impossible to optimize because of memory constraints in the computer being used to carry out the optimization process. A further advantage of the invention is the alleviation of register pressures which occur when optimizing a large program. For example, when many expressions are moved outside of loops, they exceed the number of registers available to contain them. The code which performs register allocation then requires a great deal of processing time and memory space in trying to accommodate these register requirements and may result in having to put register values into temporary storage to be retrieved later when the values are used. With partitioned optimizations in large programs, code movement of the type just described is restricted to inner sub-regions, and thus a register's assignment is placed closer to its uses. Whilst this may not be the theoretically optimal choice if an expression which is moved to the beginning of an inner loop is also a constant in the outer region or regions, it frequently avoids the wasted processing of having to reverse the effects of a previous optimization which results in poorer code quality.

APPENDIX I

```
commoning_map: proc(scr#) returns(integer);

   /*************************************************************/
   /*                                                          */
   /*  Procedure Name: commoning_map                            */
   /*                                                          */
   /*  Function: Build maps of the basic blocks and the used items */
   /*        items in the dictionary and the reg table for a    */
   /*        specific region.                                  */
   /*                                                          */
   /*  Inputs:  scr# - indicates the node in the flow graph where */
   /*             we start looking for a region for which t    */
   /*             build the reg_map and dic_map.  If start =   */
   /*             node_count then we'll try building the maps  */
   /*             for the whole program.  Otherwise, it means  */
   /*             we've already processed this region and we   */
   /*             want to find others in the program starting  */
   /*             from where we left off.                      */
   /*                                                          */
   /*  Outputs: dic_map, reg_map, map_info, node_map.          */
   /*                                                          */
   /*************************************************************/


dcl scr#        integer value;
```

```
dcl partcom      bit;
dcl (i,j,k)      integer;
dcl node         integer;
dcl limit        integer;  ! limit for vector sizes for partitioned commoning.
dcl small_limit  literally(2048);    ! 2 k
dcl medium_limit literally(1024000); ! 1 meg
dcl large_limit  literally(2048000); ! 2 meg

if indebug('PC_OFF') ¬= 0 then   ! if partitioning is turned off
  partcom = false;               ! then set the flag to false
else
  do;                            ! Otherwise, set the flag to true and
    partcom = true;              ! choose the correct data space limit.
    if indebug('PC_S') ¬= 0 then ! NOTE: the default is the medium limit.
      limit = small_limit;
    else
      if indebug('PC_L') ¬= 0 then
        limit = large_limit;
      else
        limit = medium_limit;
  end; ! else

! Allocate the arrays used to map the entities which will be collected
! by commoning.
if allocation(dic_map) = 0 then
  do; ! if dic_map isn't allocated then none of the maps are allocated.
    allocate dic_map extents(dic_tab_top);
    allocate reg_map extents(last_reg);
    allocate unmap_reg extents(last_reg-first_reg+1);
    allocate node_map extents (node_count);
```

```
  allocate unmap_node extents (node_count);
 end;  ! if allocation(dic_map) = 0

! Find a region small enough to work with...
do while (scr# ¬= 0);
 ! Reset the maps for the region
 next_reg. next_dic = 0;
 $zero(dic_map); $zero(reg_map); $zero(unmap_reg);
 $zero(node_map); $zero(unmap_node);

 ! The loop pre_header node maps into position 1 of unmap_bb and
 ! the post_exit node which is always basic block #1 maps into
 ! position 0 which is not explicitly done here since node_map is
 ! zeroed above.
 unmap_node(0) = post_exit;
 node_map(pre_header) = 1;
 unmap_node(1) = pre_header;
 node_map(post_exit) = 0;
 curr_node = 1;
 map_node(scr#);

 ! Loop thru the instructions in the nodes that have been mapped.
 do k = 1 to curr_node;  ! don't need to look at the post_exit node
                         ! since it is a "fictional" node which contains
                         ! no code and is used to simplify optimization
                         ! algorithms.
   node = unmap_node(k);
 if scr(node) then iterate;
 else
  do i = each instruction in the basic block "node".
   do j = each operand in the current instruction "i"
    if j is a register and hasn't been mapped yet then...
```

```
        next_reg = next_reg + 1;
        reg_map(j) = next_reg;
        unmap_reg(next_reg) = reg;
        end;  ! if j is a register
      else
       if j is a dictionary entry and hasn't been mapped yet then...
        next_dic = next_dic + 1;
        dic_map(j) = next_dic;
        end;  ! if j is a dictionary entry
      end do j;
    end do i;
  end do k;

! If partitioned commoning is turned off OR ELSE partitioned commoning
! IS being performed and this region ISN'T too big, then perform
! commoning on this region.
if (¬partcom) |* (¬too_big) then
  return (scr#);

! Otherwise, look for another region to map within this region.
do i = each node in the region
  if scr(i) then leave;

  end do i;

! If there are no inner regions in scr#, find the next one
! at the same or higher level.
if i > number of nodes in the region "scr#"
  scr# = find_next_region(scr#);
else ! examine the first inner region of "scr#"
  scr# = i;
end; ! do while (scr# ¬= 0)
```

```
! If we reached here, then there are no more regions which can be
! optimized.
return(0);


map_nodes: proc(scr#);
! Recursive procedure to map all the basic block and scr nodes contained
! in the region "scr#".
dcl scr#   integer value;
dcl i      integer;


! Map each node in the region "scr#".
do i = each node in the region
  ! If this node is itself a strongly connected region, then map its
  ! nodes first.
  if scr(i) then
    map_nodes(i);
  else
    do; ! Not a strongly connected region (therefore, it's a basic block)
        ! so go ahead an map it.
      curr_node = curr_node + 1;
      node_map(i) = curr_node;
      unmap_node(curr_node) = i;
    end; ! else
  end do ii;


! After mapping all of its nodes, add the "scr#" node to the
! node_map, too.
curr_node = curr_node + 1;
node_map(scr#) = curr_node;
unmap_node(curr_node) = scr#;


end; ! map_nodes
```

```
too_big: proc returns (bit);
!-----------------------------------------------------------------------
! VECTOR       VECTOR DIMENSIONS       SIZE OF A SINGLE VECTOR ELEMENT
!-----------------------------------------------------------------------
! avail_regs   next_reg x node_count    2 bytes (half a word)
! dead_regs    next_reg x node_count    2 bytes
! avail_dics   next_dic x node_count    2 bytes
! reg_kills    next_reg x next_reg      1 byte (one quarter of a word)
!-----------------------------------------------------------------------
if ( 2 * (next_reg * node_count * .5) +   ! sizes for avail_regs and dead_regs
     1 * (next_dic * node_count * .5) +   ! size for avail_dics
     1 * (next_reg * next_reg * .25)      ! size for reg_kills
   ) > limit then   ! If total amount of memory needed by these
  return(true);     ! vectors is greater than the limit then return
else                ! true because the region is too bit to be commoned.
  return(false);    ! Otherwise, return false.
end; ! too_big
find_next_region: proc(regn) returns(integer);
!-----------------------------------------------------------------------
! This procedure take a region number and finds the next region in the
! flow graph which is at the same or higher level as "regn". The node
! number of that next region will be returned. If there is no such
! region, the value 0 is returned to indicate this.
!-----------------------------------------------------------------------
dcl regn   integer value;
dcl cr     integer;        ! containing region
dcl i      integer;        ! loop counter

! We quit if we are already at the outermost region level.
do while(regn ¬= node_count);
  ! "cr" is the region which contains "regn"
  cr = containing_region(regn);
```

```
! Loop through the remaining siblings in the containing region.
do i = regn + 1 to last node in "cr"
 ! If there is another region, return its number.
  if scr(i) then return(i);
  end do i;

! At this point, we have to ascend a level and search the siblings
! of regn's containing region.
regn = cr;
end; ! do while (regn ¬= node_count);

! No more regions to examine...
return(0);

end; ! find_next_region

end commoning_map;
```

We claim:

1. A method for operating a compiler program in a computer system, said computer system utilizing said compiler program to execute a specified procedure for improving an intermediate language version of a computer program under development, comprising the steps of:

a) developing a control flowgraph representing a largest strongly connected region in said program under development;

b) by examining code sequences in said strongly connected region, determining an amount of memory of said computer system required to perform said specified procedure to said strongly connected region;

c) comparing said amount of memory determined in step b) to a predetermined computer system memory usage limit to determine if said amount of memory determined in step b) exceeds said predetermined computer system memory usage limit, thereby denoting an incapability of said computer system of applying said specified procedure to said strongly connected region;

d) if said amount of memory determined in step b) does not exceed said predetermined limit, applying said specified procedure to said strongly connected region examined in step b); otherwise

e) if said amount of memory determined in step b) exceeds said predetermined limit, executing steps b)–d) to a next largest previously unselected strongly connected region in said control flowgraph.

2. A method as claimed in claim 1, in which said specified procedure is for performing code motion in said intermediate language version of said development computer program.

3. A method as claimed in claim 1, in which said specified procedure is for performing store motion in said intermediate language version of said development computer program.

4. A method as claimed in claim 1, in which said specified procedure is for removing redundant code sequences in said intermediate language version of said development computer program.

5. A method as claimed in claim 1, in which said specified procedure is for removing useless code sequences in said intermediate language version of said development computer program.

6. A method as claimed in claim 1, in which said specified procedure is for simplifying expressions in said

intermediate language version of said development computer program.

7. A method for operating a compiler program in a computer system, said computer system utilizing said compiler program to execute a specified procedure for improving an intermediate language version of a computer program under development, comprising the steps of:

a) developing a control flowgraph representing a largest strongly connected region in said program under development, said largest strongly connected region being a parent strongly connected region and including one or more nested strongly connected regions;

b) defining said largest strongly connected region as a first strongly connected region;

c) by examining code sequences in said first strongly connected region, determining an amount of memory of said computer system required to perform said specified procedure to said first strongly connected region;

d) comparing said amount of memory determined in step c) to a predetermined computer system memory usage limit to determine if said amount of memory determined in step c) is less than said predetermined computer system memory usage limit, thereby denoting a capability of said computer system of applying said specified procedure to said first strongly connected region;

e) if said amount of memory determined in step c) is less than said predetermined limit, applying said specified procedure to said first strongly connected region examined in step c) and determining if said first strongly connected region is nested within a first parent strongly connected region in said control flowgraph, and if so;

f) determining if said first strongly connected region is nested within a first parent strongly connected region in said control flowgraph, and if so;

f) defining said first parent strongly connected region as said first strongly connected region and repeating steps c)–e); and

g) if said amount of memory determined in step c) is greater than said predetermined limit, defining a next largest previously unselected strongly connected region in said control flowgraph as said first strongly connected region and repeating steps c)–f).

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,193,190

DATED : Mar. 9, 1993

INVENTOR(S) : Joyce M. Janczyn and Peter W. Markstein

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Col. 7-8, in the table, sixth line of text, delete "which t" and substitute therefor --which to--;

Col. 15-16, in the table, 17th line of text, delete "ahead an" and substitute therefor --ahead and--;

Col. 22, line 13, delete entire line;

line 14, delete entire line; and

line 15, delete entire line.

Signed and Sealed this

Fifth Day of December, 1995

Attest:

Bruce Lehman

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

US005212794A

United States Patent [19]

Pettis et al.

[11] **Patent Number: 5,212,794**

[45] **Date of Patent: May 18, 1993**

[54] **METHOD FOR OPTIMIZING COMPUTER CODE TO PROVIDE MORE EFFICIENT EXECUTION ON COMPUTERS HAVING CACHE MEMORIES**

[75] Inventors: **Karl W. Pettis**, San Jose; **Robert C. Hansen**, Santa Clara, both of Calif.

[73] Assignee: **Hewlett-Packard Company**, Palo Alto, Calif.

[21] Appl. No.: **531,782**

[22] Filed: **Jun. 1, 1990**

[51] **Int. Cl.**[5] **G06F 9/445; G06F 9/45**

[52] **U.S. Cl.** **395/700; 364/DIG. 1;** 364/280.4; 364/280.5; 364/243.4; 364/243.41; 364/243.42; 364/261.3; 364/261.4; 364/261.5; 364/262.5; 364/264.4; 364/267.4; 371/19

[58] **Field of Search** .... 364/DIG. 1, DIG. 2; 395/700; 371/19

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,702,005	10/1972	Ingalls, Jr.	395/575
3,800,291	3/1974	Cocke et al.	395/400
4,435,758	3/1984	Lorie et al.	395/800
4,782,444	11/1988	Munshi et al.	395/700
4,847,755	7/1989	Morison et al.	395/650
4,881,170	11/1989	Morisada	395/375
4,965,724	10/1990	Utsumi et al.	395/700
4,991,080	2/1991	Emma et al.	395/375
4,991,088	2/1991	Kam	395/425
5,021,945	6/1991	Morrison et al.	395/375
5,103,394	4/1992	Blasciak	395/575
5,142,679	8/1992	Owaki et al.	395/700

OTHER PUBLICATIONS

Hwu, W. and Chang, P.; Achieving High Instruction Cache Performance with an Optimizing Compiler; 1989, pp. 242-251.

Samples A. and Hilfinger, P.; Code Reorganization for Instruction Caches; Oct. 1989, pp. 1-25.

Primary Examiner—David L. Clark

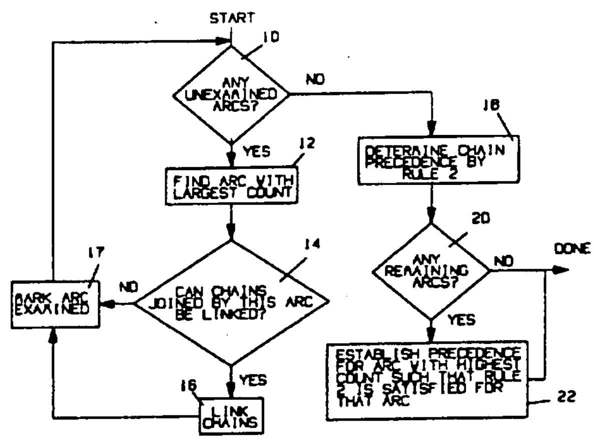
Assistant Examiner—Matthew C. Fagan

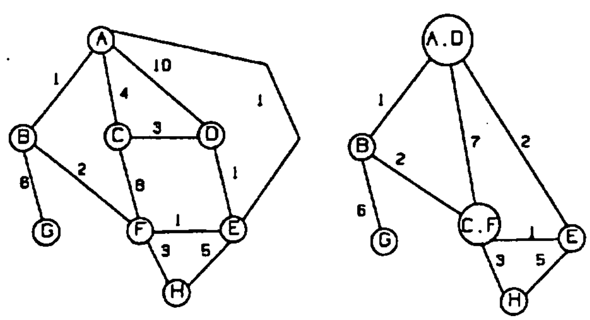
Attorney, Agent, or Firm—Alan H. Haggard; Roland I. Griffin

[57] **ABSTRACT**

The method uses statistical information obtained by running the computer code with test data to determine a new ordering for the code blocks. The new order places code blocks that are often executed after one another close to one another in the computer's memory. The method first generates chains of basic blocks, and then merges the chains. Finally, basic blocks that were not executed by the test data that was used to generate the statistical information are moved to a distant location to allow the blocks that were used to be more closely grouped together.

6 Claims, 5 Drawing Sheets





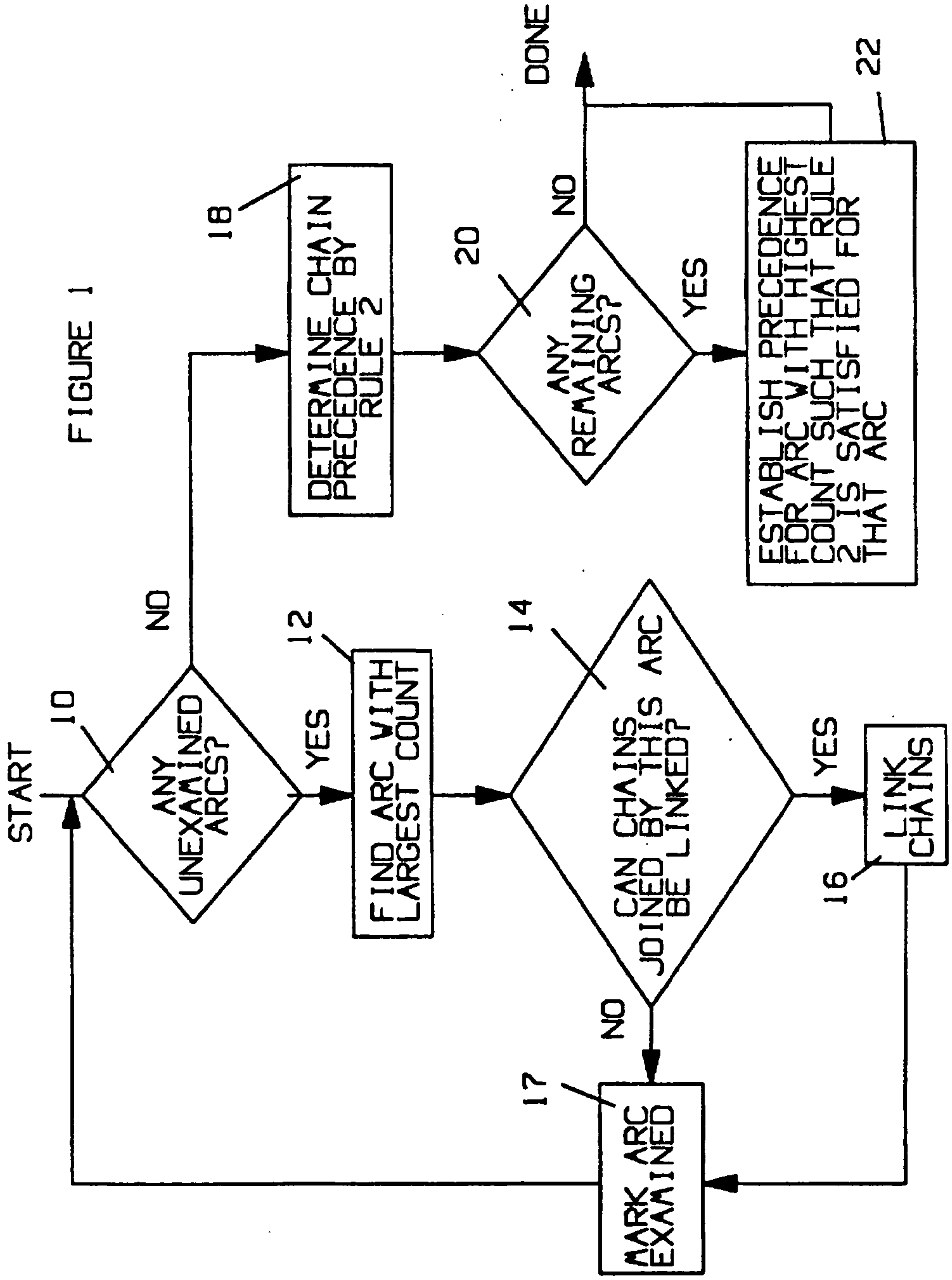
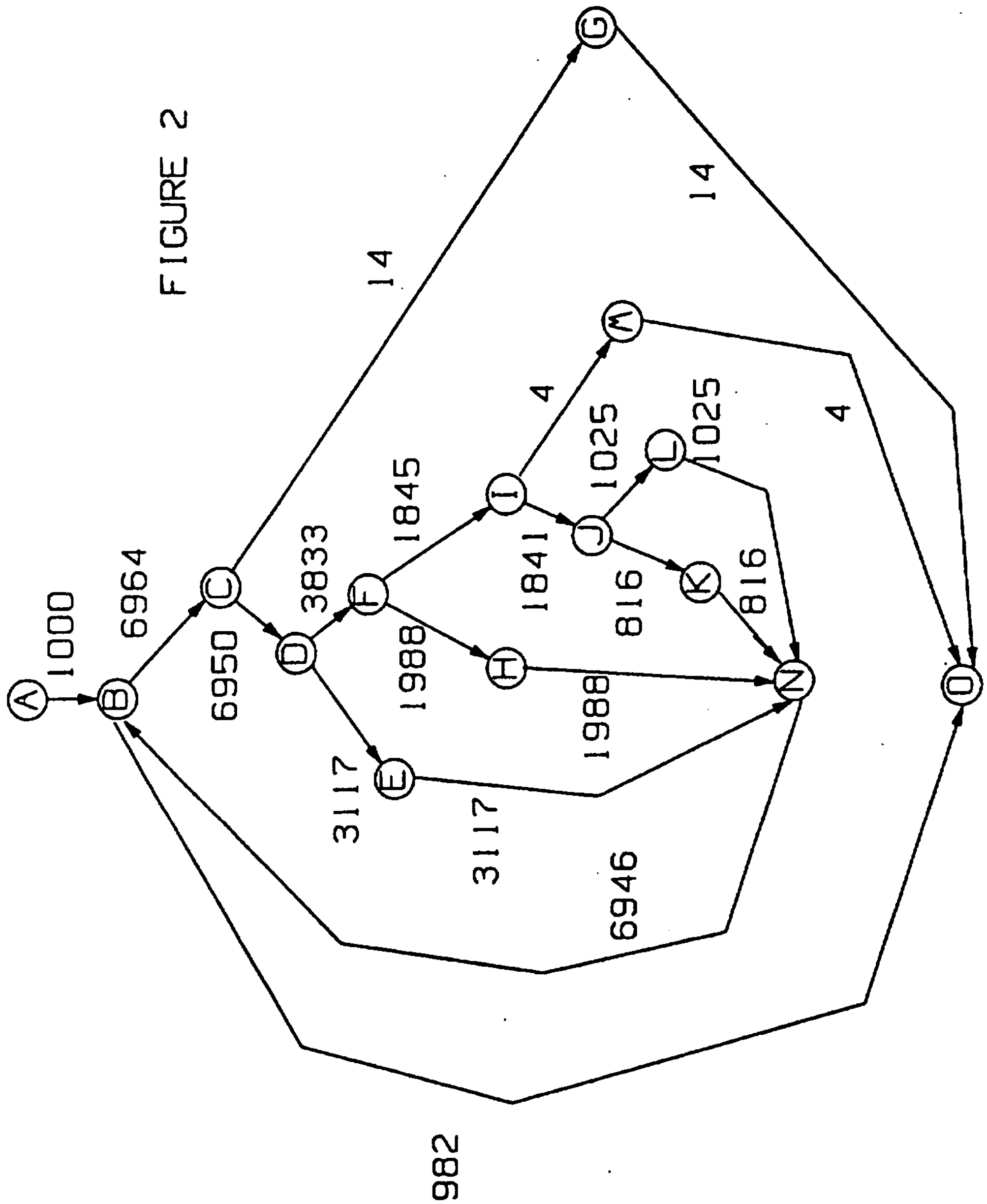

START
ANY UNEXAMINED ARCS?
10
YES
NO
FIND ARC WITH LARGEST COUNT
12
CAN CHAINS JOINED BY THIS ARC BE LINKED?
14
YES
NO
LINK CHAINS
16
MARK ARC EXAMINED
17
DETERMINE CHAIN PRECEDENCE BY RULE 2
18
ANY REMAINING ARCS?
20
NO
YES
DONE
ESTABLISH PRECEDENCE FOR ARC WITH HIGHEST COUNT SUCH THAT RULE 2 IS SATISFIED FOR THAT ARC
22

FIGURE 1

FIGURE 2


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
1000
6964
6950
3833
3117
1845
1988
14
4
1025
1841
816
6946
982

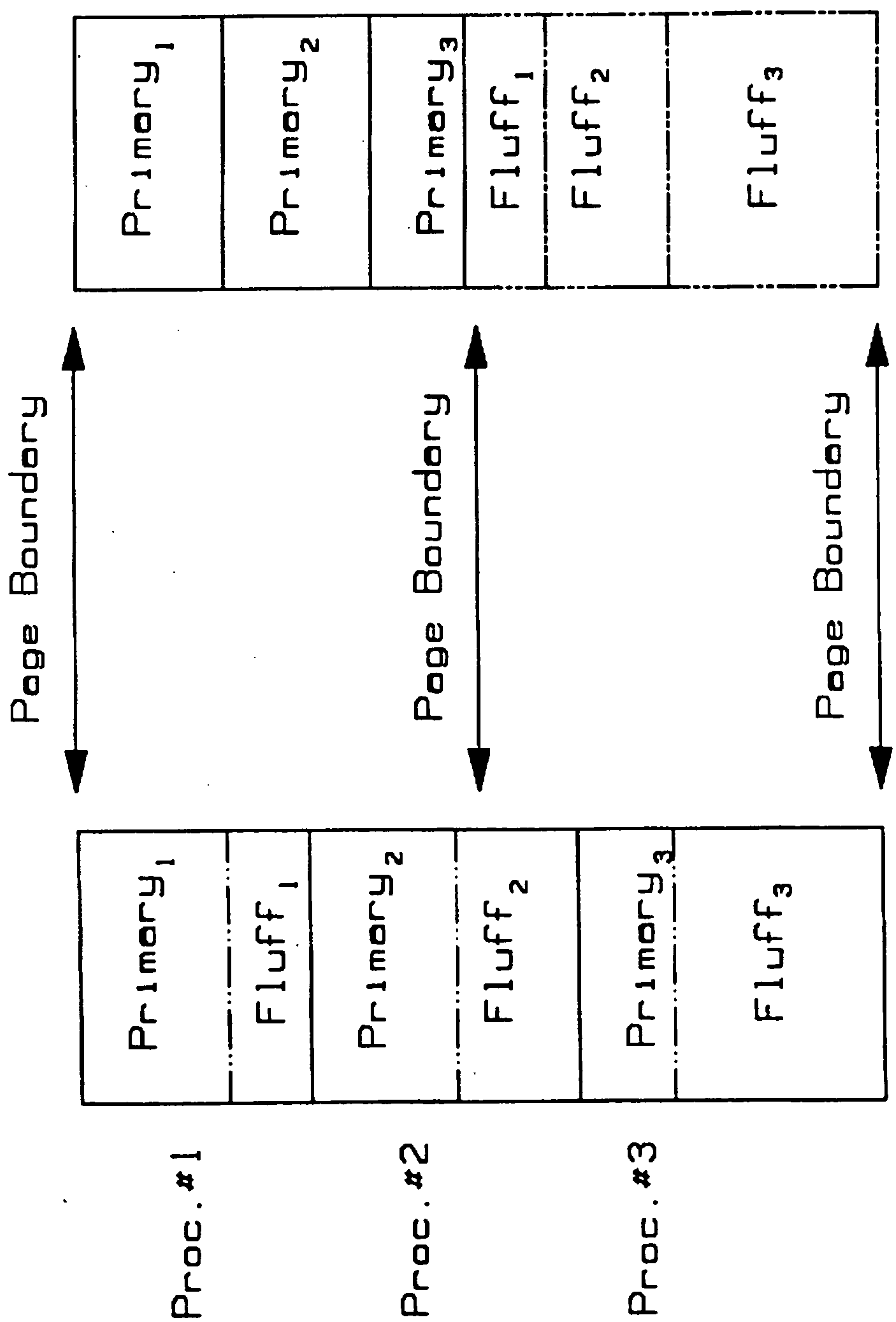

Primary$_1$
Primary$_2$
Primary$_3$
Fluff$_1$
Fluff$_2$
Fluff$_3$
Page Boundary
Page Boundary
Page Boundary
Primary$_1$
Fluff$_1$
Primary$_2$
Fluff$_2$
Primary$_3$
Fluff$_3$
Proc. #1
Proc. #2
Proc. #3

FIGURE 3

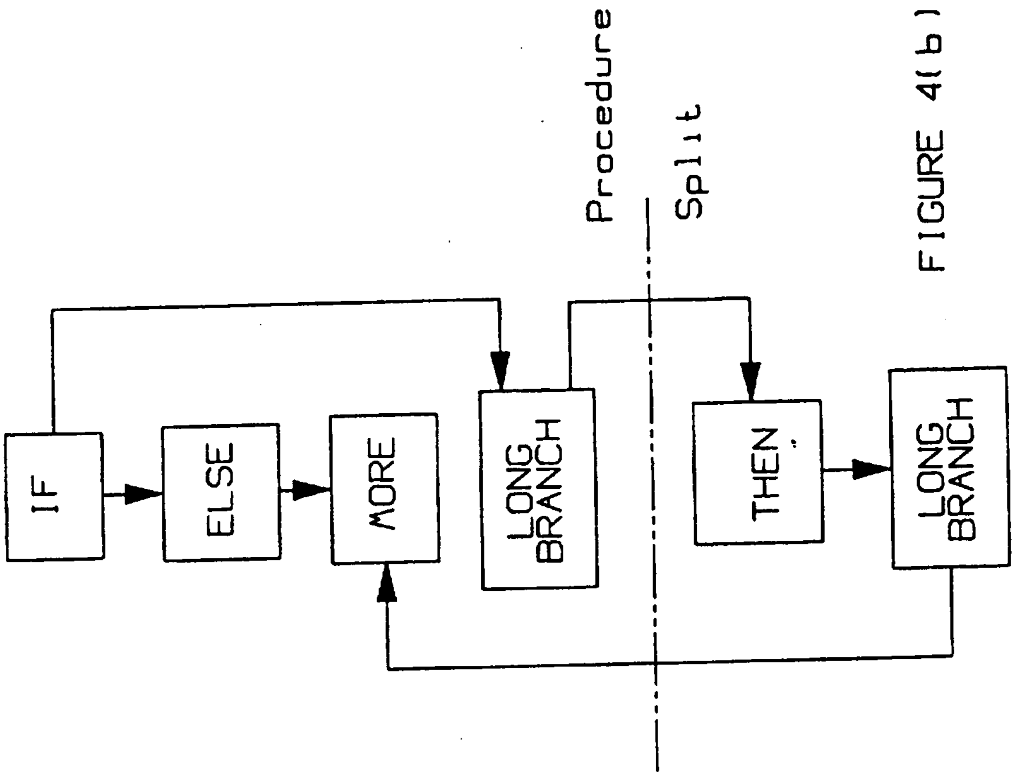


FIGURE 4(b)

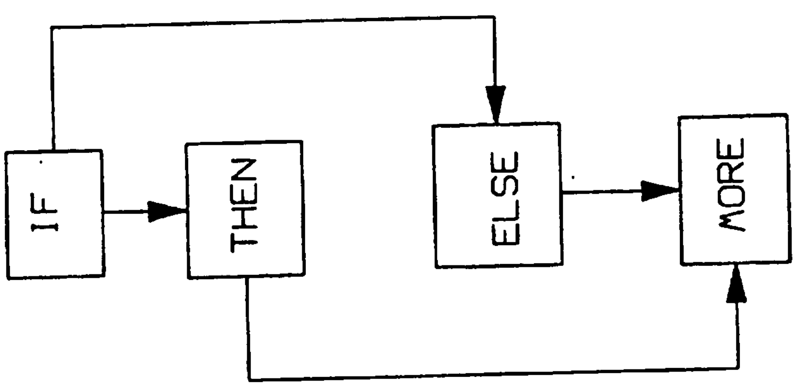


FIGURE 4(a)
(PRIOR ART)

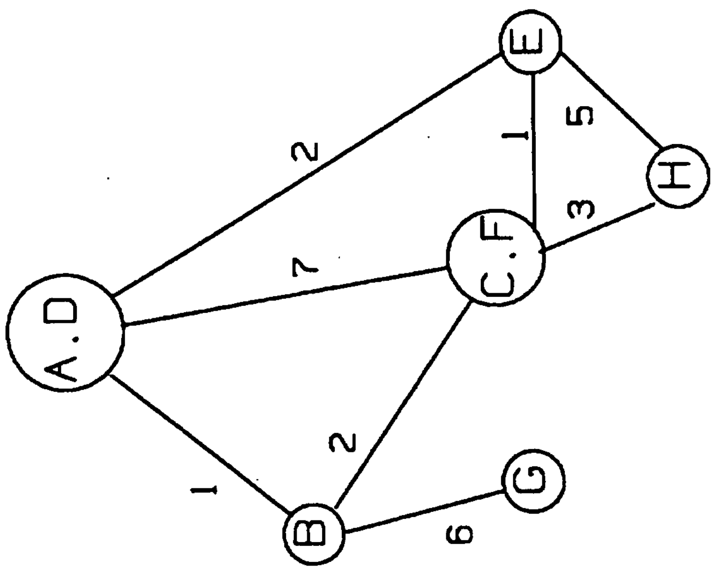


FIGURE 5(b)

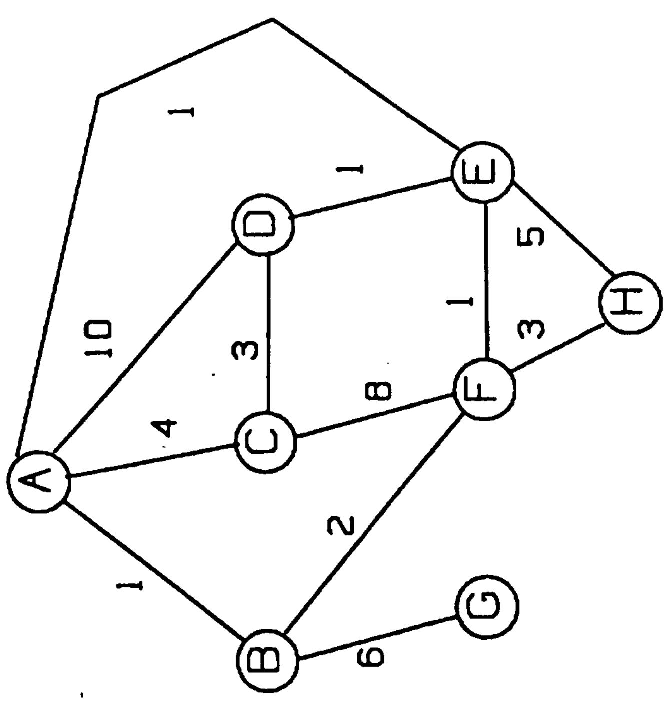


FIGURE 5(a)

METHOD FOR OPTIMIZING COMPUTER CODE TO PROVIDE MORE EFFICIENT EXECUTION ON COMPUTERS HAVING CACHE MEMORIES

BACKGROUND OF THE INVENTION

The present invention relates to computer systems and, more particularly, to an improved method for generating computer code which executes more efficiently on computers having cache memories and the like.

Conventional computing systems include a central processing unit which executes instructions which are stored in a memory. The cost of providing higher speed central processing units has decreased much faster than the cost of computer memory. This decrease in cost at the central processing level has resulted from such innovations as reduced instruction set architectures and instruction pipelining. Similar advances have not taken place in memory circuitry. In addition, the size of computer programs has increased as more complex tasks are set-up for computer implementation. This increased program size further aggravates the central processor memory mismatch.

One solution to the mismatch between central processing speed and the cost of computer memory having comparable speed is the introduction of intermediate memory units. Small high speed cache memories are placed between the central processing unit and the computer's main memory. The cache memory has a speed which is comparable to that of the central processing unit; hence, instruction fetches from the cache do not introduce processing delays. The size of the cache is much less than that of the main memory; hence, the cost of a slow main memory and a cache is much less than a main memory having the speed of the central processing unit.

When the central processing unit requests an instruction, the request is intercepted by the cache. If the instruction in question is already stored in the cache, it is returned to the central processing unit. If the instruction is not present, the cache loads a small block of memory locations which includes the requested instructions from the main memory. The cache then returns the requested instruction. The need to load the small block, referred to as a "line", delays the execution of the program; hence, such cache loads must be limited to a small fraction of the instruction fetches if the cache architecture is to provide a substantial speed enhancement. This will be the case if the computer code is organized such that the execution sequence of the code is the same as the storage sequence. That is, if instruction A is to be executed after instruction B, instruction B should be stored immediately after Instruction A. In this case, each instruction in each line of code will be executed in the order stored and cache loads will only be needed at the end of each line of code. Furthermore, the cache loads at the end of lines could then be anticipated in advance and thus performed before the lines in question were actually requested.

The presence of JUMP instructions in the computer code results in difficulties in providing this ideal code organization. When the central processing unit encounters a JUMP instructi n, the next instructi n will be at a point in the code which may be very distant from the location at which the JUMP instruction is stored. Hence, the cache may not contain the line which includes the next instruction. Furthermore, if the JUMP instruction is a conditional one, there is no method of determining in advance whether r not the jump will be executed.

JUMP instructions cause additional problems for pipelined central processing units. In a conventional pipelined processor, a number of instructions are being processed at any given time. Each time an instruction is executed, a new instruction is introduced into the "pipe". If a jump is executed, the next instruction in the pipe is not likely to be the target of the jump. Hence, a portion of the pipe must be emptied and reloaded starting with the target instruction. Hence, computer code which minimizes the use of JUMP instructions is highly desirable.

Broadly, it is the object of the present invention to provide an improved computer code optimization method

It is another object of the present invention to provide a code optimization method that places code blocks which are executed in sequence in memory locations which are close to each other.

It is yet another object of the present invention to provide a code optimization method that reduces the number of JUMP instructions in the executed computer code.

These and other objects of the present invention will become apparent to those skilled in the art from the following detailed description of the invention and the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a flow chart for a basic block positioning method according to the present invention.

FIG. 2 illustrates the basic block positioning method of the present invention with reference to a specific example.

FIG. 3 illustrates the benefits obtained by splitting procedures.

FIG. 4(*a*) shows the traditional ordering of basic blocks. FIG. 4(*b*) shows the ordering after Procedure Splitting has been applied.

FIGS. 5(*a*) and 5(*b*) illustrate the manner in which procedures are ordered according to the present invention.

SUMMARY OF THE INVENTION

The present invention comprises a code optimization method for providing more efficiently executable computer code for running on a computer system having cache memories and the like. The method uses statistical information obtained by running the computer code with test data to determine a new ordering for the code blocks. The new order places code blocks that are often executed after one another close to one another in the computer's memory.

In one embodiment of the method of the present invention a first computer program is converted to a second computer program. The first computer program comprising a set of basic blocks of computer code arranged in a first linear order, and the second computer program comprises a second set of basic blocks in a second linear rder, each basic block of said second set of basic blocks corresponding to a basic block of said first set f basic blocks. The method first determines the frequency with which each basic block of said first set of basic blocks transfers control to each other basic block f said first set of basic blocks when said first

computer program is executed with predetermined test data. An indicator is provided for each said pair of basic blocks having a determined transfer frequency greater than zero, said indicator having a first state indicating that said pair of basic blocks has not been examined and a second state indicating that said pair of basic blocks has been examined. The indicators are initialized to the first state. The pair of basic blocks for which said indicator is in said first state having the largest transfer frequency is then determined and said indicator set to said second state. A pair of chains, each comprising one or more basic blocks, is combined if the source basic block for the selected pair of chains is the tail of one of said chains and the target block is the head of the other of said chains, said source and target basic comprising said pair of basic blocks. The combining steps are repeated until all said indicators are in said second state. The chains are then arranged in a linear order to form said second computer program.

In a second embodiment of the present invention, a computer program comprising a plurality of procedures is modified to cause the computer program to be loaded in a more efficient memory order by determining the frequency with which each said procedure calls each other said procedure in said computer program and placing the procedures in the computer's memory in an determined by said determined frequencies.

DETAILED DESCRIPTION OF THE INVENTION

The present invention utilizes two passes to optimize the computer code. In the first pass the code is compiled and special code inserted which collects statistics on the subsequent execution of the code. This special code will be referred to as the instrumentation code in the following discussion. The instrumented code is then executed using representative test data. During the execution, the instrumentation code records data indicating the number of times each block transfer control to the other blocks in which various blocks of code were actually used. In the second pass, this data is used to reorder the code such that instructions that are often executed after one another are stored close to one another in memory. The reordering operation also eliminates a significant number of JUMP instructions in the portion of the code which is most frequently executed, thereby reducing the program size and further improving the efficiency with which the code is executed.

The smallest unit of computer code that is moved by the method of the present invention will be referred to as a basic block. A basic block is defined to be a straight line section of code with a single entry point at the beginning and a single exit point at the end. The basic block was chosen as the smallest entity to move because once a basic block is entered, it is guaranteed that all of the instructions for that block will be executed in order. Hence, no further improvement in cache loading can be obtained by reordering the code within the basic block.

Basic Block Positioning

Within most computer programs, there are control flow paths and basic blocks that are seldom executed during typical program runs. Conventional compilers generate code in a manner in which these rarely used sections of code are interspersed with frequently used sections. For example, consider the code for an error check in the form:

If (test for error) then (handle error case) (remaining code)

When this code is compiled, the compiler typically places the code for the error test at the end of a basic block with a conditional branch around the code to handle the unusual situation. In the normal case, there is no error, and the branch is usually taken. As a result, not all the instructions in the cache line containing the branch instruction, or the cache line containing the target, will be executed. Incompletely using some cache lines results in the total number of cache lines used being higher than would be expected from the number of instructions actually executed.

In addition, on pipelined computers, the hardware often predicts that forward conditional branches will not be taken and that backward conditional branches will be taken. For the code generated in this example, the prediction for the forward branch is usually incorrect. This leads to wasted cycles resulting from the need to reload the pipeline. The deeper the pipeline, the larger the penalty.

The present invention identifies such cases and moves the infrequently executed code away from the more frequently executed code so that the normal flow of control is in a straight-line sequence. The target and sense of the conditional branch are changed to reflect the new ordering of the code. As a result, longer sequences of code are executed before a branch is taken. Thus, on average, the number of instructions executed per cache line increases, and there are fewer caches misses. Furthermore, branch penalties are reduced since the hardware prediction matches normal code execution. Finally, in some cases, moving the infrequently executed code results in the elimination of an unconditional JUMP instruction.

When the code is instrumented in the first pass, a small block of code is added to each basic block for each possible transfer to that basic block. This code will be referred to as a basic block stub in the following discussion. When the compiler encounters a JUMP instruction, it transfers to the target of the JUMP instruction and inserts a basic block stub. The basic block stub comprises storage space for a counter used to record the number of times the jump is executed, data specifying the source of the JUMP instruction, and a JUMP instruction to the beginning of the basic block. A basic block stub is inserted for each possible JUMP instruction having the basic block in question as its target. The counters are initialized to zero by the linkage editor prior to the execution of the code with the test data.

After the code has been execute with the test data, the present invention examines the basic block stubs and constructs a graph having arcs showing the various jumps and the number of times each JUMP instruction was executed. The code is then reordered.

The reordering combines basic blocks into chains. A chain is defined to be a sequence of basic blocks that should be a contiguous section of straightline code. A source basic block is defined to be a basic block ending in a branch instruction such as a Jump. A target basic block is defined to be the basic block which begins with the target of a branch instruction. It should be noted that every basic block is a target of some other basic block, and every basic block is also a source basic block in a computer program running under a standard operating system.

Three rules are used to construct chains. First, two chains can be merged if the source basic block for the selected arc is the tail of a chain and the target block for the selected arc is the head of a chain. Since all basic blocks are themselves chains, this rule will always be true for the first arc selected.

Second, if the source of the arc is not a tail of a chain or the target of the arc is not a head of a chain, then the chains are not merged. However, such arcs are used to define a precedence relation among the chains. The precedence relationships are used to guide the final placement of the chains. When two chains cannot be merged, the chain containing the source is given precedence over the chain containing the target if the chain containing the source basic block consists of a conditional branch to a target basic block in the second chain.

If the source basic block for the current arc contains a conditional branch, then the current arc contains one of the two possible targets. If the source is not the tail of a chain, then the source has already been chained with the second of the possible targets. Since arcs are examined in the order of the frequency of use from most frequently executed to least frequently executed, the source basic block in question will have already been chained with the dominant target. On pipelined machines, the lesser taken path of a conditional branch should be a forward branch in order to avoid penalties resulting from incorrect branch prediction. Thus, since the most common target basic block for the branch has already been chained to the branch, the target for the current arc should be a forward branch from the branch point. This is the rationale behind giving the chain containing the source node precedence over the chain containing the target node

As will be discussed in detail below, the above mentioned two rules do not always resolve the ordering of all of the basic blocks. If there are any chains for which a precedence relationship has not been specified by the second rule, a third rule is applied. The third rule establishes the precedence of remaining chains by giving precedence to a chain currently lacking precedence if this would result in the second rule being satisfied for the arc having the largest count.

FIG. 1 is a flow chart for the basic block positioning method as described above. The method examines each of the arcs in the graph. If any arc has not been examined, the method finds the arc having the largest count among the unexamined arcs as shown at 10 and 12. The method then attempts to join the two chains connected by this arc as shown at 14. If the two chains can be linked then they are linked as shown at 16. The arc is then removed. If the two chains can not be linked, the arc is merely marked as being examined and the next unexamined arc is found.

When all of the arcs have been examined, the program applies the second rule to establish precedence between any remaining chains as shown at 18. If there are any remaining arcs, the method then attempts to remove them by satisfying the second rule for the arc having the largest count.

This reordering operation may be more easily understood by considering a simple example. FIG. 2 shows the basic block graph of a procedure with each arc labeled with its count as determined in a test run. The source code was a while loop written in the C language. Within the body of the loop were several if-then-else tests along with two alternate break statements at basic blocks G and M. The reordering starts by finding the arc with the highest count which is the arc from basic block B to C (count =6964) and joining these two together into a chain. The next largest arc (6950) from basic block C to D is then found. Since basic block C is at the end of a chain, the B-C chain can be extended to include D. The next three largest arcs (N-B, D-F, and E-N) also extend the existing chain. The next largest arc (D-E) can not be used to extend a chain, since basic block D is already in the middle of a chain. The process is repeated with the remainder of the arcs. The final result will be six chains, three of which being individual basic blocks:

1) A
2) E-N-B-C-D-F-H
3) I-J-L
4) G-O
5) K
6) M

After the chains have been formed, a precedence relation is attempted between them based on the desire to have non-taken conditional branches be forward. There are 6 conditional branches in this procedure. The second rule discussed above prescribes that chain 2 containing B-C should occur before chain 4 containing 0. If these two chains are placed in this order, then the less frequently executed path out of B is the forward branch as desired.

The other conditional branches at C,F, I and J imply that the following order among the chains:

chain 2 (C)	before	chain 4 (G)
chain 2 (F)	before	chain 3 (I)
chain 3 (I)	before	chain 6 (M)
chain 3 (J)	before	chain 5 (K)

There still remains one conditional branch left at D. Hence, the second rule will not always provide a complete set of precedence relationships. As noted above, it is preferred that the branch which is less frequently taken be forward. Hence, the branch from D to E should be forward (since that link is not taken as often as the path from D to F). But since E and D are part of the same chain, and E occurs before D in the chain, that branch is necessarily backwards. In other examples, it sometimes happens that the precedence relation between chains is not transitive. In these cases the reordering method resolves such conflicts by looking at the weights of the arc and choosing an order that will satisfy the precedence of the arcs having the higher weight.

After the precedence criteria are determined between chains, the chains are arranged so that the order is satisfied consistent with the above rules. If there is some freedom in choosing the next chain, the chain connected to existing chains by the heaviest count is chosen. In this example, the final order of the basic blocks is:

A, E-N-B-C-D-F-H, I-J-L, G-0, K, M

An alternative ordering method which is less optimum then the one described above may also be used. In the alternative method, the chains are started with the entry point basic block. At each point, the computer tries to place a successor of the last placed basic block.

If there is more than one available, the one with the higher arc count is chosen. In this example, the method starts with basic block A. It only has one successor, B, so that would be placed next. B has two successors, but C is connected with the higher arc count so it would be chosen next. Continuing in this way, blocks D, F, H and N are placed.

After N is chosen, its only successor has already been selected, so a new starting place must be chosen from among the unselected basic blocks. In this example, E is selected since it is connected to the already chosen blocks by a weight of 3117. Note that its only successor has also already been chosen; hence, another new starting point is needed. The computer starts again with an unselected block, I. The final order after the algorithm completes is A, B, C, D, F, H, N, E, I, J, L, 0, K, G, and M.

Both of the above methods do a good job of making conditional branches into forward branches that are usually not taken. However, the first method has the added advantage of removing unconditional branches. Hence, the first method is the preferred embodiment of the present invention.

After a new ordering of the basic blocks has been established, the frequently executed basic blocks (primary basic blocks) are now found toward the top of the procedure while the infrequently executed blocks are found near the end. Basic blocks that were never executed according to the collected profiling data will be referred to as fluff. The fluff ends up at the very end of the procedure.

Although this ordering and rearrangement of the code is a significant improvement over that generated by the compiler, there is still room for further improvement. In particular, the code would execute more efficiently if the fluff were moved to a distant location thereby allowing all of the primary basic blocks to be grouped together. This separation process is referred to as procedure splitting since it results in part of the code in a given procedure being relocated to a remote area of memory.

Procedure Splitting

Procedure Splitting is the process of separating the fluff basic blocks of a procedure into a separate secondary region of the main memory to minimize the size of the primary part of the procedures. By producing a smaller and denser primary procedure, more primary parts of procedures can now be packed into a single memory page. Hence a reduction in the working page set size is often obtained. This is a significant advantage of the present invention on computers which utilize virtual memory systems.

Virtual memory systems are analogous to the main memory-cache memory arrangement discussed above. In this case, the slow memory usually consists of a disk drive, and the fast memory consists of the computers main memory. Large programs are stored on the disk. As the instructions are needed, a portion of the program, referred to as a page, is read from the disk into the main memory of the computer. A page, in turn, consists of a plurality of lines which are read into the cache memory prior to execution by the central processing system. By reducing the number of pages needed to store the commonly used code, the number of times the desired page is not already in memory when requested is reduced. By moving the fluff, fewer pages are needed to hold the working sections of the program.

FIG. 3 shows an example of three procedures that would normally require two pages of memory. By splitting the primary portions away from the fluff, the primary code for the procedures can be contained within a single page.

When separating the fluff basic blocks from the primary blocks for a given procedure, a new procedure is defined to encompass the fluff blocks. This procedure is atypical in that it does not adhere to the standard procedure calling conventions. There are no defined entry or exit points and no register saves or restores. This method was selected to avoid the overhead associated with a standard procedure call whenever control transfers between a primary and fluff procedure. The new procedure is also given a higher sort key so that the linker will force all fluff procedures to the end of the program space.

After it has been determined which basic blocks will be moved into the fluff procedure, any branch that goes from a primary basic block to a fluff basic block, or vice-versa, is redirected to a long branch. A "long branch" is a JUMP instruction whose target may be more than a predetermined distance from the JUMP. Long branch instructions take longer to execute, and hence, are normally avoided. However, these instructions are less of a problem in the present invention, since they are only rarely executed. The target of the long branch will be the original target basic block. A long branch is used because eventually the fluff procedure will most likely be located too far from its primary counterpart to be accessed by a short branch instruction.

These branches are allocated at the end of the procedure that is doing the interprocedural branch. Placing the branch at the end of the procedure instead of converting the original branches into long branches was chosen for simplicity. If there are multiple branch sites which have the same branch target, a single long branch is shared to save space.

Since the paths between primary and fluff code were never taken according to the representative profiling data, the long branches should be infrequently executed and only for atypical inputs.

An example of an if-then-else sequence where the then clause is considered to be fluff is shown in FIG. 4(*a*)-(*b*). FIG. 4(*a*) shows the traditional ordering of basic blocks. FIG. 4(*b*) shows the ordering after Procedure Splitting has been applied.

The procedure splitting method of the present invention has been described with respect to basic blocks which were not utilized by the program while running the test data. However, it will be apparent to those skilled in the art that any basic block which had a frequency of utilization less than some predetermined frequency could be moved in the same manner.

Procedure Ordering

In the preferred embodiment of the present invention, yet another layer of code optimization is utilized. The above described basic block reorganization is preferably performed on basic blocks within a single procedure. Most computer programs consist of a plurality of procedures. Hence, in the preferred embodiment of the present invention, the reordering process is also performed at the procedure level.

This separation is made because the compiler normally has access to the basic blocks within a procedure but can only be assumed to have the code for a single procedure available at any given time. The linkage

editor, on the other hand, has access to the code generated by the compiler for all of the procedures, but does not have the code for basic blocks in a form which is convenient for inserting the instrumentation code.

In the preferred embodiment of the present invention, the procedure ordering is implemented in a separate set of passes. The first step in ordering the procedures is to construct a weighted call graph using the data collected by the instrumentation code. The linker introduces the procedure instrumentation code in a manner analogous to that described above for the basic blocks. The instrumentation is setup by the linker because only the linker is assured of having access to all procedures at the same time. The basic instrumentation code is similar to that described above with reference to the basic block, and hence, will not be described in detail here. For the purposes of this discussion it is sufficient to note that a table is setup having a counter for each combination of calling and called procedure as is done for source and target basis blocks. Since the linkage editor already has a table giving the location of all calls and the procedure called, this is a straight forward extension of a conventional linkage editor. This table is initialized by the linker prior to running the program with the sample test data.

After running the program, a call graph is constructed. Initially, each node of the graph is a single procedure and the edges correspond to calls between the procedures. The edges are weighted by the number of times the calls were actually made. If a procedure calls another from several different places within itself, or if two procedures were mutually recursive, those weights are first summed together to form the edge weight shown in the graph.

Broadly, the method of the present invention places procedures in memory based on the frequency with which said procedures called each other when executing the test data. The linkage placement of the procedures is determined by constructing groups similar to the chains described above with reference to basic blocks. However, the rules for ordering of the groups are somewhat different. Initially each group consists of one of the nodes in the call graph. The link order is constructed by first choosing the edge with the heaviest weight. The two nodes connected by this edge will be placed next to each other in the final link order to form a new group. However, the precedence of one node over the other in the group is not determined at this stage. This node will be referred to as an incompletely determined node. The two nodes are merged into a single node or group having two procedures therein, and the remaining edges leaving each node are coalesced.

This process is repeated until the two nodes being combined include at least one node that consists of an incompletely determined node. At this point, the nodes are combined, and the order of the various procedures in the node is determined. The order is determined by first making a list of the possible orders. Those entries in the list that place procedures next to one another that were not linked in the original call graph are then removed from the list. If there is more than one possible order left, the one which places procedures closest to each other that had the highest count connecting them in the call graph is used. Once the order has been determined, the linked procedures in the node are treated as a single "procedure" or an incompletely determined node in the remaining steps. The process is repeated until the resulting graph consists of an individual node or nodes with no edges.

An example of this process is shown in FIG. 5. FIG. 5(*a*) shows the graph prior to any mergers. The edge with the highest weight is the one connecting procedures A and D. Hence, procedures A and D will be placed next to each other in the final link order. However, the precedence of A and D will not be decided until later in the process. The two nodes are then merged and the process repeated. The next edge will be the one between procedures C and F. Again, it is determined that C and F will be placed next to one another in the final link order; although it is not known which will come first. After nodes C and F are merged, the graph will appear as shown in FIG. 5(*b*).

The next highest weight is that between the two nodes that were just built by the previous mergers. Since it has already been decided that A and D should be together and that C and F should be together, there are four distinct choices available for ordering of this next merger:

(1) A-D-C-F or F-C-D-A
(2) A-D-F-C or C-F-D-A
(3) D-A-C-F or F-C-A-D
(4) D-A-F-C or C-F-A-D

The second order in each case is merely the reverse of the first order. To determine which of these to choose, the original connections between the procedures are examined. First, the choices involving connections that did not exist in the original graph are eliminated. In particular, procedure F is connected neither to procedure A nor to procedure D in the original graph. Hence, possibilities (2) and (4) are eliminated. If there are any remaining choices, preference is given to the order that reflects the strength of the original ordering. For example, procedure C was more strongly connected to procedure A (weight of 4) in the original graph than to procedure D (weight of 3). Hence, it is preferred that C and A be adjacent in the final order, as long as the already determined groupings can be satisfied. Hence, the grouping D-A-C-F or its equivalent grouping F-C-A-D is preferred.

This process continues until the graph has no edges left. If the graph was a connected graph, then the final step will lead to a single node. If the graph was disjoint, then there will be several independent nodes. This later case can occur when a procedure is not called at all for the test data used to generate the graph.

There has been described herein a method for optimizing computer code. Various modifications to the present invention will become apparent to those skilled in the art from the foregoing description and accompanying drawings. Accordingly, the present invention is to be limited solely by the scope of the following claims.

What is claimed is:

1. A method for operating a digital computer, including a memory for the storage of computer instructions and data, to convert a first computer program to a second computer program, said first computer program comprising a set of basic blocks of computer code arranged in a first linear order and said second computer program comprising a second set of basic blocks in a second linear order, each basic block of said second set of basic blocks being derived from one of said basic blocks of said first set of basic blocks, said method comprising the steps of:

(a) determining the frequency with which each basic block of said first set of basic blocks transfers con-

trol to each other basic block of said first set of basic blocks when said first computer program is executed with test data;

(b) providing an indicator for each pair of said basic blocks of said first set of basic blocks having a determined transfer frequency greater than zero, each said indicator having a first state and a second state, each said indicator being initialized to said first state;

(c) determining which pair of basic blocks of said first set of basic blocks for which said indicator corresponding to said pair is in said first state has the largest transfer frequency and setting said indicator corresponding to said pair to said second state;

(d) combining a pair of chains of said first set of basic blocks to form a new chain which replaces said pair of chains if a source basic block for said pair of chains is the tail of one of said chains and a target basic block is the head of the other of said chains, said source basic block and target basic block being said pair of basic blocks determined in step (c);

(e) repeating steps (c) and (d) until all said indicators are in said second state; and

(f) arranging said chains in a linear order to generate said second computer program.

2. The method of claim 1 wherein said step of arranging said chains in a linear order further comprises the step of placing a first one of said chains in front of a second one of said chains if said first chain contains the source of a conditional branch and said second chain contains the target of said conditional branch.

3. The method of claim 2 wherein said digital computer further comprises means for loading said second computer program into said computer memory at memory locations specified by labels in said second computer program and wherein said step of arranging said chains further comprises the step of placing the target basic block of each pair of chains combined in step (d) having a determined transfer frequency less than a predetermined value into a code segment separate from that containing the source basic block of said each pair of chains combined in step (d), and labeling said separate code segment in a manner that will cause said separate code segment to be placed in a specified area of computer memory when said second computer program is loaded into said digital computer prior to execution of said second computer program.

4. The method of claim 3 further comprising the step of converting a branch instruction ending said source basic block of one of said chains to a long branch instruction.

5. In a digital computer system including a computer memory, a method for causing said digital computer system to load a computer program comprising a plurality of procedures, wherein at least some of said procedures when executed by said computer system call others of said procedures, into said computer memory, said method comprising the steps of:

(a) determining the frequency with which each procedure calls each other procedure in said computer program; and

(b) placing said procedures in said computer's memory in an order that depends on said determined frequencies,

wherein said step of placing said procedures comprises the steps of:

(c) defining groups of said procedures, each group initially comprising one of said procedures, each group being defined as being determined or incompletely determined, said groups being initially defined to be incompletely determined;

(d) defining a weight for each pair of said groups, said weight initially being the determined frequency with which said procedure in one group of said each pair called said procedure in the other group of said each pair;

(e) combining a pair of said groups having the highest weight to form a new group which replaces each group in said pair of groups;

(f) determining an order for the procedures in said new group if one of the groups in said combined pair was defined as being incompletely determined, said new group being defined as being determined if an order is determined for all procedures in said new group, and being defined as incompletely determined otherwise;

(g) defining weight for each pair of groups containing said new group, is said weight being the sum of the frequencies with which said new group called the other group in said each pair of groups; and

(h) repeating steps (e) through (g) until only one group remains, the order in which said procedures are loaded into said computer memory being determined by the order of said procedures in said remaining group.

6. The method of claim 5 wherein said step of determining an order for the procedures in a group which resulted from the combination of two groups in which one of said groups was defined as being incompletely determined and in which said combined groups includes at least three said procedures, said step of determining an order comprises the steps of:

constructing a list of all possible orderings of procedures in said new group group;

eliminating orderings in which a pair of procedures in which neither procedure calls the other are placed next to each other; and

choosing one of said remaining orderings as the order for the procedures in said new group.

* * * * *